View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Digital library of Brno University of Technology

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PŘÍKLADY METAPROGRAMOVÁNÍ V C++

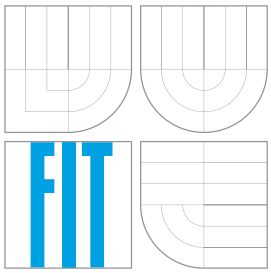
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ KUKLÍNEK

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PŘÍKLADY METAPROGRAMOVÁNÍ V C++

C++ METAPROGRAMMING EXAMPLES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ KUKLÍNEK

VEDOUCÍ PRÁCE
SUPERVISOR

Dr. Ing. PETR PERINGER

BRNO 2010

Abstrakt

Tato práce pojednává o metaprogramování v jazyce C++. Obsahuje přehledovou část zaměřenou na metaprogramování obecně a dále sadu příkladů demonstrujících různé techniky podporující metaprogramování v C++ s důrazem na nové vlastnosti nadcházející verze normy, zatím zvané C++0x. Příklady ukazují použití šablon s proměnným počtem parametrů, jejich výpočetní sílu, staticky polymorfní generování kódu a také obecnou implementaci několika návrhových vzorů.

Abstract

In this thesis we investigate possibilities of metaprogramming in C++. It contains a general overview of metaprogramming and a set of examples of various metaprogramming techniques in the C++ programming language with an emphasis on the possibilities proposed by the upcoming standard, called C++0x for now. Examples demonstrate usage of variadic templates, computational power of templates, statically polymorphic generation of a runtime code and generic implementation of several design patterns.

Klíčová slova

metaprogramování, C++, šablony, objektově orientované programování, návrhové vzory

Keywords

metaprogramming, C++, templates, object oriented programming, design patterns

Citace

Lukáš Kuklínek: Příklady metaprogramování v C++, bakalářská práce, Brno, FIT VUT v Brně, 2010

Příklady metaprogramování v C++

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Dr. Ing. Petra Peringera

.....
Lukáš Kuklínek
17. května 2010

Poděkování

Zde chtěl bych poděkovat svému vedoucímu Dr. Ing. Petru Peringerovi za vstřícný přístup, pomoc a množství užitečných připomínek během psaní práce.

© Lukáš Kuklínek, 2010.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

Cílem této práce je poskytnout ucelený přehled metaprogramování, nástrojů, které jej využívají či podporují, a pojmenovat hlavní přednosti i slabiny tohoto přístupu. Metaprogramování je zvláštní technika vývoje softwaru. Dokáže značně zlepšit znovupoužitelnost kódu, zvýšit úroveň abstrakce a zrychlit vývoj aplikace. Stejně tak však může vést i k značnému zatemnění implementace a dlouhým hodinám stráveným nad laděním příliš komplexního systému nadmíru obecných modulů.

Hlavní část práce představuje sbírka demonstračních příkladů využití metaprogramování v C++. Tento programovací jazyk byl zvolen proto, že se jedná o jediný široce používaný jazyk s úspěšnou implementací vestavěné podpory pro metaprogramovací techniky pomocí šablon.

Právě zavedení klíčového slova *template*, které přineslo obohacení jazyka C++ o generické programování, znamenalo průlom, kdy z jednoho šablonového kusu kódu mohla vzniknout ne jedna, ale přímo několik tříd, funkcí či metod v cílovém programu. Netrvalo dlouho a zvídaví počítačoví nadšenci začali zkoumat, čeho všeho se s šablonami dá dosáhnout, jak daleko se až dá zajít. Jejich objevy nepřestávají udivovat kohokoli, kdo se pokusí do problematiky šablon proniknout hlouběji.

Jelikož se v těchto dnech, týdnech a měsících právě vyvíjí nová verze normy jazyka C++ a překladače již některé nové vlastnosti experimentálně implementují, nemůže tato zůstat bez povšimnutí. Právě naopak se budeme na příkladech snažit demonstrovat i sílu a možné využití nových vlastností C++0x, jak se tato verze standardu označuje.

Práce předpokládá znalost základů jazyka C++. Znalost šablon alespoň na úrovni používání standardní knihovny také nebude na škodu.

Kapitola 2

Metaprogramování

2.1 Co je to metaprogramování?

> *Metaprogramování* znamená psaní kódu (programu), jehož výstupem je program či modifikace programu (případně i jeho samého), nebo který provede část práce v době překladu, ačkoli by ji běžně provedl až za běhu. [26]

Důvodů, proč se vývojář uchýlí k využití metaprogramování, může být několik. Naskýtá se vidina vyšší produktivity a obecná neochota psát opakující se části kódu ručně, zvýšení flexibility a obecnosti řešení pro možné budoucí použití, či zlepšení efektivity výsledného programu tím, že se některé úlohy přesunou z výsledného programu na program, který jej vygeneroval.

Princip však zůstává stejný. Program není cílovým produktem, nýbrž pouze prostředkem k jeho vytvoření. Nazveme jej tedy *metaprogramem* a jazyk, ve kterém je napsán, označíme jako *metajazyk*. *Cílový jazyk* je takový, se kterým metaprogram manipuluje nebo je výstupem metaprogramu. Cílový jazyk nemusí být nutně čitelný člověkem, může se jednat o vnitřní reprezentaci metastruktur programovacího jazyka či strojový kód.

Speciálním případem metaprogramování je i psaní překladače. Například překladač jazyka Pascal je metaprogramem s jazykem C jakožto metajazykem. Manipuluje se strukturami jazyka Pascal a jeho výstupem je strojově čitelný kód.

Rozlišujeme metaprogramování *statické* a *dynamické*. Statické metaprogramování se děje v době překladu, dynamické až za běhu programu.

2.2 Generování kódu

Nyní se posuneme od vytyčení pojmů na trochu konkrétnější úroveň. Podíváme se, jakými různými způsoby se dá metaprogramování realizovat bez přímé podpory v programovacím jazyce. Generování kódu taktéž představuje nejjednodušší a nejpřímočařejší způsob metaprogramování.

Program po obdržení odpovídajícího vstupu vypíše výsledný zdrojový kód v cílovém jazyce. Pro mnohé specifické a často se opakující úlohy z oblasti informačních technologií již existují dobře zavedené nástroje. Vše bude podloženo příklady z praxe.

2.2.1 Preprocesory

Myšlenkou preprocesoru je nahradit určité části vstupu jinými nebo vstup nějak obohatit.

Generické preprocesory Provádí textové nahrazení maker za jejich definici.

Typickým zástupcem je makroprocesor GNU M4. Uživatelé Linuxu a podobných systémů, kteří se někdy pokusili zkompilovat nějaký software ze zdrojových kódu, museli pravděpodobně použít skript `./configure`. Vězte, že tento skript byl vygenerován právě zpracováním vstupních definic pomocí M4.

Vývojová prostředí V záplavě všemožných integrovaných vývojových prostředí se často vyskytuje funkce „ústřižků kódu“ s možností nahrazení speciálních značek (třeba ve tvaru `<#ClassName#>`) případ od případu pomocí formuláře. Jedná se v podstatě o adaptaci výše uvedeného principu preprocesoru na grafické rozhraní.

Aspect weavers Aspektově orientované programování adresuje problémy prostupující naskrz celým kódem. Umožňuje například přidat nějakou metodu do každé třídy splňující daná kritéria. Programy, které provádí tuto transformaci zvané *weavers*, jsou často implementovány právě jako preprocesory [17]. Více o souvislosti mezi AOP a metaprogramováním se dočtete na straně 7.

2.2.2 Generátory kódu

Na rozdíl od preprocesorů, vstup generátorů kódu se zásadně liší od jejich výstupu. Následující výčet není ani zdaleka úplný, ale může pomoci udělat si obrázek o rozsahu a rozmanitosti problematiky generování kódu.

UML Nástroje podporující tento modelovací jazyk umí vygenerovat zdrojové soubory s kostrou tříd rozdělených do modulů podle grafického návrhu.

Qt Meta-Object Compiler Vstupem pro Meta-Object Compiler z toolkitu pro tvorbu grafických uživatelských rozhraní Qt je hlavičkový soubor. Pokud obsahuje třídy s makrem `Q_OBJECT`, vygeneruje compiler nový hlavičkový a zdrojový soubor. Výsledkem je mimo jiné obohacení schopností jazyka C++ o mechanismus signálů a slotů a přidání omezené schopnosti reflexe za běhu programu.

Generátory parserů Řeší jeden z nejběžnějších úkolů programátora: stavbu syntaktického analyzátoru. Typicky přijímají pravidla často ve formátu podobném BNF a po jejich zpracování máme k dispozici zdrojový kód obsahující implementaci parseru popsaného danými pravidly. Mezi zástupce této skupiny nástrojů se řadí například systémy *Yacc*, jeho GNU protějšek *Bison*, nebo *XPL* [13].

Rozhraní pro jiné jazyky Existují nástroje, které z rozhraní tříd a procedur jednoho programovacího jazyka vygenerují rozhraní pro jazyk jiný. Jako příklad uveďme *SWIG*.

Vzdálené volání procedur Často používaná technika v systémech jako *D-Bus* či *RPC* [16]. Vstupem je definice rozhraní. Výstupem pak implementace v daném jazyce, která se postará o to, aby se volání provedlo vzdáleně, přestože se programátorovi jeví jako lokální.

2.3 Programovací jazyky s podporou metaprogramování

Nyní si posvítíme na programovací jazyky, které podporu pro metaprogramování přímo či nepřímo zahrnují. Také pojmenujeme související paradigmata.

Jazyk C Vestavěný preprocesor dává jazyku C jistou možnost metaprogramování. Je třeba přitom použít maker se všemi jejich výhodami i slabinami a zrádnými zákoutími.

Dokonce i řádně ozávorkované makro `MAX(a, b) ((a) > (b) ? (a) : (b))` se nevyhne opakovanému vyhodnocení argumentů včetně případných nechtěných vedlejších efektů. Navíc nemůžeme dopředu říct, který argument se vyhodnotí dvakrát a který jen jednou.

Po zasazení do naší terminologie je cílovým jazykem jazyk C a metajazyk představují direktivy preprocesoru.

Příklad Výhodou preprocesoru jazyka C je jeho schopnost manipulovat s identifikátory. Napíšeme tedy makro `CONCAT` spojující dva symboly dohromady:

Ukázka 2.1: Makro CONCAT

```
1 #define CONCAT_(A, B) A ## B
2 #define CONCAT(A, B) CONCAT_(A, B)
```

Pokusíme se s jeho využitím vyrobit generický kontejner podobným stylem, jako bychom v C++ použili šablony. Půjde o abstraktní datový typ zásobník. Pro parametrizaci typu jednotlivých položek v zásobníku použijeme makro `TYPE`. Nejdříve definujeme rozhraní v souboru `stack.inc.h`, jak vidíme v ukázce 2.2.

Ukázka 2.2: Rozhraní ADT zásobník v jazyce C

```
1 #define STACKNAME CONCAT(stack_,TYPE)
2 /* forward declaration */
3 struct STACKNAME;
4 /* functions */
5 struct STACKNAME* CONCAT(STACKNAME,_new)(void);
6 void CONCAT(STACKNAME,_push)(struct STACKNAME* stack, TYPE element);
7 void CONCAT(STACKNAME,_pop)(struct STACKNAME* stack);
8 /* more functions ... */
```

Obdobně vytvoříme implementaci a uložíme jako `stack.inc.c`. Zájemci o bližší prozkoumání nahlédnou do přiložených zdrojových kódů.

Nyní zbývá vytvořit instance naší „pseudošablony“ pro datový typ `char*`.

Ukázka 2.3: Vytvoření instancí zásobníku v jazyce C

```
1 // mystacks.h - interface of my stack instantiations
2 typedef char *cstr; // '*' cannot be a part of an identifier, so use typedef
3 #define TYPE cstr
4 #include "xmacro/stack.inc.h"
5 // mystacks.c - implementation of my stack instantiations
6 #include "mystacks.h"
7 #define TYPE cstr
8 #include "xmacro/stack.inc.c"
```

Od nynějška máme k dispozici rodinu funkcí pro manipulaci se zásobníkem C-čkovských řetězců ve tvaru `stack_cstr_push()` a podobně. Neztrácíme přitom typovou kontrolu.

Stejně tak bychom mohli vytvořit zásobníky pro další datové typy. Pro techniku (jejíž slabý odvar je výše uvedený příklad) založenou na kombinaci vzájemně vnořených maker se vžilo označení *X-Macros* [14].

2.3.1 Šablonové metaprogramování

Během této metaprogramovací techniky překladač zpracuje bloky kódu zvané *šablony* k vygenerování dočasného mezikódu. Ten je následně spojen s nešablonovým kódem a přeložen.

Výsledkem zpracování šablon bývají datové typy (v C++ struktury nebo třídy), konstanty či funkce. Snad mi čtenář odpustí, když už teď prozradím, že si v kapitole 4.2.4 ukážeme techniku, jak v C++ napsat šablonu, která vyprodukuje jinou šablonu.

Šablony jsou součástí mnoha programovacích jazyků. Jmenujme například C++, D, Eiffel, Haskell (s rozšířením) či ML. Na některé z nich se nyní podíváme blíže.

Jazyk C++

Šablony v jazyce C++ představují jedinou masivně rozšířenou formu šablonového metaprogramování. Datové typy, celočíselné konstanty a jiné šablony mohou vystupovat jako parametry šablon.

Jelikož je tato problematika podrobně rozebrána ve zbytku práce, nebudeme se jí zde dále zabývat a zájemce odkážeme rovnou na třetí kapitolu (strana 8).

Jazyk D

Jazyk D svým pojetím staví na jazyce C++, svým návrhem se snaží pozvednout jeho silné a potlačit slabé stránky. Jde však o zcela nový programovací jazyk nezatížený dědictvím jazyka C.

Disponuje propracovaným systémem šablon. Syntaxe je poněkud příznivější, než je tomu v případě C++. Parametrem šablony může být i identifikátor (*alias*). Syntaxe jazyka také přímo umožňuje nastavit jednotlivým šablonám omezující podmínky (*constraints*).

Jazyk D se pyšní také silnou podporou pro práci s jednoduchými i složenými datovými typy v době překladu. Taktéž umožňuje nevytvářet šablonu celé třídy nebo algoritmu, ale pouze její část a poté její instanci na příslušné místo vložit (*mixin*).

Na závěr malá ukázka výpočtu hashe v době překladu z oficiálního webu [4]:

Ukázka 2.4: Jednoduchá šablona v jazyce D

```
1 template hash(char [] s, uint sofar=0)
2 {
3     static if (s.length == 0) const hash = sofar;
4     else const hash = hash!(s[1 .. length], sofar * 11 + s[0]);
5 }
6 const uint hello_world_hash = hash!("hello world");
```

Hlavní nevýhodou oproti C++ je především menší rozšířenost a z toho plynoucí nižší počet knihoven a podpůrných nástrojů. Nová norma C++0x také snižuje náskok tohoto jazyka co do celkového návrhu.

Java

Zde si dovolím malé upozornění. Ačkoli se koncept *generics* v jazyce Java svou syntaxí velmi podobá šablonám z C++, **nejedná** se v tomto případě o metaprogramování [12].

V Javě tedy například `ArrayList<String>` a `ArrayList<Double>` ve zkompilované podobě sdílejí stejný bytecode a jsou oba ekvivalentní k `ArrayList<Object>`.

Typový parametr je použit pouze ke kontrole datových typů za překladu. Tím pádem se zvyšuje výkonnost a předchází nečekaným programátorským chybám.

2.3.2 Aspektově orientované programování

AOP (aspektově orientované programování, [10]) se dá při troše dobré vůle považovat také za techniku metaprogramování. Programátor píše definice, jejichž efekty budou po překladu prostoupeny celým kódem. AOP je spíše doplňkem jiného paradigmatu, typicky objektově orientovaného programování, než samotné plnohodnotné paradigma.

Problémům, jejichž řešení se v neaspektových programovacích jazycích jen těžko koncentruje na jedno místo v kódu, říkáme *cross-cutting concerns*. Právě tento typ problémů AOP adresuje. Může jít například o ladění, logování, serializaci či synchronizaci vláken. Šablony tedy dovolují parametrizovat lokálně, zatímco aspekty globálně.

2.3.3 Dynamické jazyky

Prakticky jakýkoli programovací jazyk disponující nějakou formou funkce `eval`, která přijmá textový řetězec a následně jej interpretuje, tímto otevírá bránu k metaprogramování. Sice se použití takovéto funkce nepovažuje obecně za dobrou praktiku, ale to nic nemění na tom, že příležitosti k metaprogramování, které nabízí, jsou prakticky neomezené.

Reflexivní programovací jazyky mohou být samy sobě metajazykem. Další stupínek představuje zboření hranice mezi daty, zápisem programu a jeho vnitřní reprezentací.

Například programovací jazyk LISP je založen na manipulaci se seznamy a také jeho zdrojový kód se zapisuje jako seznam. Můžeme tedy za běhu vytvořit seznam reprezentující nějaký algoritmus a ten následně spustit.

Kapitola 3

Metaprogramování v C++

3.1 Vývoj normy jazyka C++

Konečně se dostáváme k C++. Začneme troškou historie [18, 20].

3.1.1 Doba předstandardová

Počátky jazyka C++ se datují na rok 1979, kdy jej navrhl a vytvořil jeho počáteční implementaci dánský vědec na poli informačních technologií Bjarne Stroustrup. Tehdy se jazyk jmenoval *C With Classes*. Název docela přesně vystihuje, jaké vlastnosti obsahoval.

Jméno C++ získal jazyk v roce 1983. První verze se překládaly nástrojem `cfront`. Neprodukoval nativní aplikaci, nýbrž zdrojový kód v C, který se následně přeložil kompilátorem jazyka C.

C++ v1.0 První kulatá verze specifikace jazyka z roku 1985 přinesla krom podpory virtuálních funkcí také vlastnosti, které se v budoucnu budou hodit i pro metaprogramování. Přibyly totiž konstanty, reference a možnost přetěžovat funkce.

C++ v2.0 Verze číslo dvě (1989) obohatila jazyk o vícenásobnou dědičnost, abstraktní třídy, statické a konstantní členy tříd (funkce a proměnné) a nový kvalifikátor přístupu *protected*. Šablony stále ještě chybí.

Poté vychází dokument *The Annotated C++ Reference Manual* [6]. Vzniká již ve spolupráci se standardizační komisí. Přidává krom podpory pro výjimky, jmenné prostory a dynamickou identifikaci typu také veledůležité šablony. Za další milník můžeme považovat rok 1994, kdy se součástí specifikace stává standardní knihovna šablon (*STL*).

3.1.2 C++98

První verze posvěcená mezinárodním standardem *ISO/IEC 14882:1998* se neformálně označuje jako C++98. Obsahuje všechny výše zmíněné vlastnosti. Nutno podotknout, že standardizace trvala zhruba deset let.

V roce 2003 vyšel další standard. Opravuje některé chyby, podivnosti a inkonzistence. Honosí se předvídatelným označením C++03.

3.1.3 C++0x

Nadcházející verze normy přinese změny, které se přes konzervativní ráz standardizační komise ani v nejmenším nedají označit za kosmetické. Díky průtahům je již jasné, že nebude dosaženo původního časového cíle a poslední dvě číslice roku vydání nové normy do šablony 0x nezapadnou. Během psaní této práce vyšel final draft *ISO C++ 2010* [5, 22]. Pojďme se nyní blíže podívat, jaké novinky programátorům nabízí.

Šablony s proměnným počtem parametrů

Variadické šablony (anglicky *variadic templates*) budou pro účel této práce nejzásadnější změnou. Umožňují jednomu parametru šablony reprezentovat několik argumentů [8].

Vše se děje pomocí operátoru tři tečky (`...`), který slouží k zabalení či rozbalení parametrů na daném místě. Počet prvků se dá zjistit pomocí operátoru (`sizeof...`).

Použití bude demonstrováno v další sekci a v mnoha z příkladů. Zatím si ukážeme pouze malou lehce vykonstruovanou ochutnávku. Vytvoříme třídu, která dědí od továrny na výrobu arbitrárního počtu datových typů. Předpokládáme, že šablonu `Factory` máme k dispozici.

Ukázka 3.1: Ukázka variadických šablon

```
1 template <typename... Types>
2 class MultiFactory : public Factory<Types>... {
3     public: static const int number_of_supported_types = sizeof...(Types);
4 };
5 // a factory capable of creating integers, strings, and submarines
6 typedef MultiFactory<int, std::string, Submarine> MyCrazyFactory;
```

R-value references

Odkaz na r-hodnotu, jak by se dala tato vlastnost volně přeložit, je zvláštní typ reference. Používá se především v případě, kdy se právě vznikající objekt odkazuje na data právě zanikajícího objektu a přebírá za ně zodpovědnost. Předejde se tak zbytečnému kopírování a následnému mazání dat, například při předávání návratových hodnot z funkce [3].

Syntakticky se značí jako `Typ&&`.

Příklad Nejspíše nebude od věci vysvětlit *r-value references* trochu názorněji. Mějme funkci `f()` vracející řetězec a řetězcovou proměnnou `str`. Co provede výraz `str = f()`?

Naše funkce patrně vrací nějakou svoji lokální řetězcovou proměnnou. Nemůže ji vrátit referencí, protože bude nemilosrdně zlikvidována destruktorem, jakmile opustí tělo funkce. Je ji třeba nakopírovat do naší proměnné `str`. Tím se mimo jiné i zlikviduje původní obsah proměnné `str`. Nakonec zanikne i vnitřní proměnná v naší funkci, což znamená dealokaci zduplikovaných dat. Bilance: jedna alokace, dvě dealokace a jedno kopírování (vše relativně náročné operace).

R-value reference jsou většinou (avšak ne nutně) implementovány jako prohození obsahů dvou proměnných, v našem případě vnitřní proměnné funkce a proměnné `str`. Přitom není třeba prohazovat celé obsahy řetězců, stačí vyměnit ukazatele. Tím pádem se při likvidaci vnitřní proměnné funkce zlikvidují původní data naší proměnné `str`, k čemuž by muselo dojít tak či tak. Bilance: jedna dealokace a víceméně nenáročná operace výměny prvků.

Sjednocený způsob inicializace

Další krok k naplnění jedné ze základních tezí C++: podporovat uživatelsky definované typy stejně dobře jako vestavěné. Pomocí typu ze standardní knihovny `std::initializer_list` je možné inicializovat například vektor stejným způsobem, který byl doposud povolen pouze u vestavěných datových typů. Navíc novou syntaxi využívající složené závorky je nyní možno využít téměř všude.

Ukázka 3.2: Inicializace v C++0x

```
1     double nums[] = { 1.0, 2.2, 5.8 };                    // OK
2     std::vector<double> vec = { 1.0, 2.2, 5.8 };          // OK since C++0x
3     std::pair<int, std::string> mypair = { 5, "beer" }; // OK since C++0x
```

Uživatelsky definované literály

Ještě jeden doklad o snaze dodržet zásadu podporovat uživatelské typy stejně jako vestavěné [19]. Umožňuje uživateli nadefinovat vlastní typ literálu podobným způsobem, jako se přetěžují operátory.

Literál je možno definovat funkcí, která bere jako parametr řetězec či číslo nebo šablonovou funkcí, kterážto dostane jednotlivé znaky literálu jako šablonové parametry. Konstanty pak bude možno použít například následovně:

Ukázka 3.3: Ukázka uživatelských literálů

```
1 inline RegExp operator "" re(const char* reg) { return RegExp(reg); }
2 // use:
3 RegexMatch(some_string, "^[0-9]+%[a-z\.-]{3,10}"re);
4 Complex number = 3 + 5j;
```

Další drobné změny

V následujících odstavcích budou stručně popsány některé další novinky, které však buď nemají pro naše účely příliš velký význam nebo nejde o příliš hluboké zásahy do jazyka.

Dedukce typu Klíčové slovo `auto` získalo nový význam. Pokud jej uvedeme místo jména datového typu při inicializaci proměnné, bude její typ odvozen z výrazu na pravé straně. Komplexní číslo z předešlého příkladu bychom tedy mohli zapsat jako:

Ukázka 3.4: Ukázka nového využití klíčového slova auto

```
1 auto complex_number = 3 + 5j;
```

Jde o velmi příjemné rozšíření, zvláště pokud si uvědomíme, že názvy typů instancí složitých šablon mohou narůst vskutku obludných rozměrů.

Lambda funktory Rozšíření jazyka, které dokáže vytvořit funkční objekt včetně případného uchování kontextu přímo v toku kódu (například uvnitř funkce).

Ukázka 3.5: Ukázka použití lambda funkce

```
1 double sum_sq = 0.0; // sum of squares of list elements will be stored here
2 std::for_each(list.begin(), list.end(),
3     [&sum_sq](double x){ sum_sq += x*x; } // this is a C++0x lambda functor
4 );
```

decltype Nová konstrukce slouží k získání typu výrazu. Lze ji použít kdekoli syntaxe jazyka C++ očekává datový typ. Příklad je součástí ukázky 3.6.

Nový způsob zápisu funkcí Hodí se ve spojení s `decltype` v případě, že explicitně neznáme návratový typ funkce. Místo návratového typu se napíše `auto` a samotný výsledný typ se zapíše až za seznam parametrů následovaný „šipkou“.

Tuto syntaxi k zápisu návratového typu je nutné použít i u lambda funktorů, které obsahují více než jeden příkaz.

Ukázka 3.6: Ukázka použití decltype a nového zápisu funkce

```
1 template <typename T>
2 auto multiply(const T& a, const T& b) -> decltype(a * b) {
3     return a * b;
4 }
```

Extern templates Explicitní instanciaci šablony je nyní možné uvodit klíčovým slovem `extern`. Tím dáváme překladači signál, že šablonu nemusí instanciovat, protože se tak stane v jiné překladové jednotce. Následkem bude zrychlení překladu z důvodu eliminace duplicitních instanciací šablon.

Úhlová závorka Odstranění nepříjemného faktu, že dva znaky ‘>’ po sobě dříve představovaly v jakémkoli kontextu operátor bitového posunu. Nová norma nám však dovoluje zapsat i následující konstrukci s očekávaným výsledkem.

Ukázka 3.7: Ukázka použití úhlové závorky

```
1 // here, '>>' is not a bit shift operator
2 std::vector<std::pair<int, double>> container;
```

Konstantní výrazy Nové klíčové slovo `constexpr` způsobí, že daný výraz se vyhodnotí v době překladu. Může jít o konstantu, ale i o konstruktor či funkci, avšak zde jsou nastavena tvrdá omezení. Konstruktor musí mít prázdné tělo, funkce musí mít tělo ve tvaru `return výraz;`. Výraz může volat pouze další constexpr funkce či odkazovat constexpr proměnné.

Konstantní výraz lze použít všude, kde překladač očekává konstantu známou v době překladu. Mezi taková místa patří počet prvků statického pole či parametry šablon. Konstantní výrazy nejsou výpočetně úplné, protože nedovolují použití smyčky a rekurze.

Smyčka založená na konceptu rozsahu Obdoba konstrukce *foreach* v jiných jazycích. Syntaxe: `for (variable : container) DoSomething(variable);`.

Novinky ve standardní knihovně Standardní knihovna byla též značně rozšířena. Namátkou jmenujme například regulární výrazy, n-tice (*tuple*), hashtabulky, generování náhodných čísel, podporu větvení vláken a mnoho dalšího.

Stejně tak jsou existující části knihovny upraveny tak, aby využívaly výše popsané nové vlastnosti.

3.1.4 Budoucnost

Tím však snahy udělat z C++ lepší jazyk zdaleka nekončí. Některé vlastnosti se plánují do budoucna, některé měly být již součástí této normy, avšak byly pro svůj rozsah a komplexnost odloženy. Jsou evidovány například návrhy pro přidání statické reflexivity či garbage collectoru.

Moduly Moduly by měly odstranit bolest způsobenou hlavičkovými soubory. Ta je ztělesněna především duplikací kódu (hlavička + implementace) a také nízkou rychlostí kompilace, kdy je překladač přes několik úrovní direktivy `#include` nucen zpracovat často i několik megabajtů kódu [24].

Koncepty Koncepty měly přinést revoluci v metaprogramování pomocí šablon, avšak jejich schvalování bylo odloženo a do C++0x se nedostaly [21]. Koncept představuje množinu požadavků na datový typ. Tím dostáváme do ruky nástroj pro „meta-typovou“ kontrolu [7]. Cílem je zpřístupnit metaprogramování a nabídnout nástroj ke specifikování požadavků na datový typ. To pomůže překladači hlásit lepší chybové zprávy (typ x nesplňuje požadavek y). Konstrukce `concept_map` měla umožnit explicitně a transparentně adaptovat datové typy na daný koncept, aniž by se změnilo jeho rozhraní. *Axiomy* měly zase umožnit překladači přijmout nějaké předpoklady o datových typech modelujících daný koncept, například pro účely optimalizací.

Experimentální implementaci konceptů nabízí projekt ConceptGCC[1]. Jak bude popsáno v sekci 3.3.1, koncepty představují důležitou abstrakci i bez explicitní podpory v programovacím jazyce.

3.1.5 Podpora nové normy

Překladač GCC již dnes podporuje slušnou várku vlastností C++0x. Podpora v dalších nástrojích, jako jsou vývojová prostředí, indexery souborů a generátory dokumentace, se však zatím jeví jako velmi slabá. Více se problematikou podpory nové normy ze strany překladačů zabývá kapitola 5.

[1] http://www.generic-programming.org/software/ConceptGCC/

3.2 Šablony

Přichází velká chvíle: napíšeme si první šablonovou funkci. Avšak nejdříve si řekenme obecně, jak se takové šablony zapisují.

Každá šablona začíná klíčovým slovem template následovaným seznamem parametrů v úhlových závorkách. Pokud je šablonovým parametrem datový typ, uvedeme jej klíčovým slovem `typename` nebo `class`. Konvence radí používat druhou z jmenovaných možností v případě, že argumentem musí být třída či struktura. Pokud může být argumentem i primitivní datový typ, použijeme variantu první.

3.2.1 Šablony funkcí

Ukažme si tedy výše popsanou syntax na konkrétním příkladu. Napíšeme funkci `max`, vracející větší ze dvou prvků.

Bude schopná pracovat s jakýmkoli typem, u kterého je možno vytvořit novou instanci kopií a implementuje operátor větší než. Jinými slovy modeluje koncepty *CopyConstructable* a *GreaterThanComparable*.

Ukázka 3.8: Šablona max

```
1 template <typename T>
2 T max(T a, T b) { return a > b ? a : b; }
```

Ukázka 3.9: Použití šablony max

```
1 max(3, 5);            // calls max<int>
2 max(3.0, 5);          // ERROR: what shall I call?
3 max<double>(3.0, 5); // explicit template argument
```

V šabloně v ukázce 3.8 vidíme typový parametr `T`, jehož výskyty jsou v prvním případě z ukázky 3.9 nahrazeny typem `int`. Také je možno vypozorovat další příjemnou vlastnost šablonových funkcí. Pokud je možné argumenty šablony vydedukovat z typů argumentů funkce, tak se tak stane. Druhý řádek skončí chybou, protože naše funkce očekává v obou parametrech výraz stejného datového typu. To je napraveno na řádku třetím uvedením šablonových argumentů explicitně.

3.2.2 Specializace

Specializace znamená vytvoření jiné implementace šablony pro určitou konkrétní množinu šablonových parametrů. Bohužel šablony funkcí (narozdíl od tříd) podporují pouze úplnou specializaci.

Představme si nyní, že chceme pomocí naší šablony `max` vracet (abecedně) větší ze dvou řetězců tak, jak s nimi pracuje jazyk C, tedy ukazatel na sekvenci znaků zakončenou binární nulou. Očividně máme problém, protože naše šablona bude porovnávat ukazatele neboli umístnění řetězců v paměti. Specializace nabízí řešení, můžeme totiž napsat jinou implementaci naší šablony pro typ `const char*`.

Ukázka 3.10: Specializace šablony max pro řetězce

```
1 template <>
2 const char* max(const char* a, const char* b) {
3     return strcmp(a, b) > 0 ? a : b;
4 }
```

Co kdybychom ale chtěli speciální šablonu pro jakékoli ukazatele s tím, že by měla porovnávat jejich dereferencované hodnoty[2]? Vytvoříme *novou* šablonu a dedukci, která se použije necháme na schopnosti překladače řešit volání přetížených funkcí. Tím se nám podařilo nahradit mechanismus částečné specializace, kterým pro funkce C++ nedisponuje.

Ukázka 3.11: Částečná specializace pomocí přetěžování funkce

```
template <typename T>
T* max(T* a, T* b) {
    assert(a != 0); assert(b != 0);
    return *a > *b ? a : b;
}
```

3.2.3 Instanciace

Instancí šablony nazveme konkrétní datový typ či funkci vzniklou dosazením parametrů do šablony. Vytvoření instance šablony se nazývá *instanciace* (někdy také *konkretizace*). Většinou se děje automaticky při jejím prvním použití – volání šablonové funkce či vytvoření proměnné šablonové třídy. Je-li třeba, můžeme si instanciaci vynutit.

Ukázka 3.12: Explicitní instanciace šablony max

```
template int max<int>(int, int);
template const char* max<const char*>(const char*, const char*);
```

Pokud bychom výše uvedené řádky začali klíčovým slovem `extern`, instanciaci potlačíme s tím, že překladač bude předpokládat, že bude mít příslušnou instanci šablony k disposici v jiné překladové jednotce a příslušný kód s ní spojí až při linkování. Vlastně bychom vytvořili dopřednou deklaraci šablonové instance.

3.2.4 Šablony tříd

Šablony tříd jsou ještě zajímavější. Asi nejpřímočařejší využití předtavuje vytvoření generických kontejnerů.

Ukázka 3.13: Šablona Array

```
template <typename T>
class Array {
public:
    typedef T member_type;                          // member type
    Array(const Array& other);                      // copy constructor
    Array(Array&& other);                           // move constructor
    Array(std::initializer_list<T> init_members); // initializer constructor
    ~Array();                                       // destructor
    T& operator[](size_t index);                    // member access
    size_t size() const { return member_num; }      // get member count
    // and so on...
private:
    size_t member_num, alloc_num; // member count and allocated size
    T* members;                   // pointer to actual data
};

template Array<char>; // explicit instantiation
```

[2]Moudrost takovéhoto rozhodnutí je sice pochybná, nicméně jako příklad nám poslouží dobře.

Implementaci členských funkcí můžeme napsat buď přímo do třídy (pak se budou chovat, jako by byly `inline`), nebo zvlášť a to následovně:

Ukázka 3.14: Implementace přístupu k prvku v šabloně Array

```
template <typename T>
T& Array<T>::operator[](size_t index) { return members[index]; }
```

Specializace

Šablony tříd (potažmo struktur) podporují částečnou i úplnou specializaci. Například šablona `std::vector<T>` ze standardní knihovny poskytuje specializaci pro datový typ `bool`. Mění implementaci tak, že nevytváří skutečné pole proměnných typu bool, nýbrž ukládá jednotlivé položky po bitech.

Takovou úplnou specializaci bychom pro naši šablonu zapsali následovně:

Ukázka 3.15: Specializace šablony Array pro typ bool

```
template <>
class Array<bool> { typedef bool member_type; /* class body ... */ };
```

Částečná specializace je možná také. Pokud bychom chtěli nějakým speciálním způsobem ošetřit zpracování ukazatelů (třeba znemožnit vložit do našeho pole nulový ukazatel), specializaci bychom provedli takto:

Ukázka 3.16: Specializace šablony Array pro ukazatele

```
template <typename T>
class Array<T*> { typedef T* member_type; /* class body ... */ };
```

Za povšimnutí stojí, že v tomto případě se liší parametr šablony `T` a její argument. V šablonové třídě `Array<int*>` zastupuje `T` z naší specializace typ `int`, nikoli `int*` [19].

3.2.5 Parametry šablon

Jak již bylo několikrát zmíněno a demonstrováno, parametry šablon bývají povětšinou datové typy. Mohou jimi však být i vestavěné celočíselné typy (například `int`), dále ukazatele či reference. Naopak parametrem šablony se nemůže stát typ operující v plovoucí desetinné čárce. Toto omezení je dnes považováno za historické [23] a přebytečné, technologicky reálným šablonovým parametrům nic nebrání.

Třetím a poslením typem parametrů šablon jsou samotné šablony. Uvozují se klíčovým slovem `template` a je třeba uvést i výčet parametrů.

Ukázka 3.17: Ukázka šablonového parametru šablony

```
template <template<typename> class Container, typename MemberType>
struct PairContainer {
    typedef Container<std::pair<int, MemberType>> type;
};

typedef PairContainer<Array, std::string>::type MyContainer;
// MyContainer is Array<std::pair<int, std::string>>
```

3.2.6 Name lookup

Mějme šablonu s parametrem `X`. Existuje členský atribut `X::mem`. Co je tento atribut zač? Je to datový typ, šablona, nebo členská proměnná? Překladač si tyto otázky klade taktéž a pokud jde o typ či šablonu, musíme mu napovědět příslušným klíčovým slovem, jak ukazuje následující kousek kódu:

Ukázka 3.18: Ukázka kvalifikátorů jmen

```
template <class X>
struct {
    typedef typename X::mem1 some_type;              // mem1 is a type
    typedef X::template mem2<some_type> other_type; // mem2 is a template
    auto func() -> decltype(X::mem3()) {             // mem3 is a function
        return X::mem3() + X::mem4;                  // mem4 is a variable
    }
};
```

S dohledáním jmen souvisí také tzv. *two-phase lookup* [9]. Symboly, jejichž význam závisí na šablonovém parametru jsou překladačem řešeny v době instanciace šablony a mohou vyžadovat specifikátory, jak demonstruje ukázka 3.18. Symboly nezávislé na šablonovém parametru jsou vyřešeny, jakmile překladač narazí na definici příslušné šablony.

Nekvalifikované symboly se implicitně považují za nezávislé, takže výraz `f()` v šabloně nemusí najít funkci `f`, kterou třída zdědila od svého rodiče. Jako řešení se nabízí udělat z identifikátoru funkce závislý symbol `this->f()`.

3.2.7 Výpočetní model

Nyní zkombinujeme naše znalosti o parametrech šablon a specializaci s faktem, že šablony můžeme instanciovat i rekurzivně. Z toho se již dá vytušit, že systém šablon v C++ je výpočetně úplný [25].

Příklad Pokusíme se donutit překladač C++, aby nám spočítal funkci, kterou bychom za běžných okolností řešili iteračně. Použijeme k tomu příklad faktoriálu. Rovnice 3.1 ukazuje jeho rekurzivní definici.

$$x! = \begin{cases} 1 & \text{pro } x = 0 \\ x \cdot (x-1)! & \text{pro } x \geq 1 \end{cases} \tag{3.1}$$

Teď tuto definici vyjádříme v řeči šablon. Budeme potřebovat šablonu s jedním celočíselným parametrem. Koncovou podmínku pro $x = 0$ realizujeme pomocí specializace.

Ukázka 3.19: Faktoriál

```
template <unsigned x>
struct Factorial {
    enum { value = x * Factorial<x-1>::value };
};

template <> struct Factorial<0> { enum { value = 1 }; };
```

Nyní stačí pro výpočet 5! v době překladu napsat `Factorial<5>::value`.

Také krásně vidíme, že jazyk šablon je deklarativní a nápadně se podobá jazykům funkcionálním. Tato podobnost ještě více vynikne, jakmile se dostaneme k práci se seznamy.

3.3 Čas překladu a čas běhu programu

Z předchozího výkladu plyne, že C++ v sobě vlastně obsahuje dva vzájemně se prolínající a doplňující jazyky. Nazvěme je třeba šablonová a nešablonová část jazyka. Říkáme, že C++ je *dvojúrovňový jazyk* [11].

Šablonová část je spuštěna již během překladu a jejím výstupem je vnitřní reprezentace nešablonového kódu. Slouží nám jako metajazyk. Nešablonová část je přeložena do strojově čitelného kódu a spuštěna až se spuštěním onoho kódu. Přehledně to shrnuje následující tabulka.

Tabulka 3.1: Srovnání metajazyka a jazyka C++

Vlastnost	**Nešablonové konstukce**	**Šablonové konstrukce**
Použití	jazyk	metajazyk
Paradigma	imperativní, objektové	deklarativní, funkcionální
Data	proměnné, objekty	datové typy, konstanty
Metadata	datové typy	koncepty
Transformace dat	funkce, metody	šablony
Generování metadat	pomocí šablon	šablony, concept_map
Znovupoužitelnost	na úrovni obejktového kódu	na úrovni syntaxe jazyka

3.3.1 Koncepty

Důležitým pojmem pro metaprogramování v C++ je koncept. Představuje seznam požadavků na rozhraní daného datového typu, které musí splňovat, aby jej bylo možno s úspěchem použít v parametru šablony. Co obsatarává na úrovni proměnných a funkcí typová kontrola, to má na úrovni šablon na starosti kontrola konceptů.

I když jazyk koncepty explicitně nepodporuje a při kontole šablonových parametrů uplatňuje *duck typing*, je dobré se s tímto pojmem seznámit a u složitějších šablon požadované koncepty řádně zdokumentovat.

3.4 Nevýhody a omezení šablon v C++

Metaprogramování pomocí šablon s sebou přináší také mnohá úskalí.

- Šablony nemají příliš přívětivou syntax. Množství „čtyřteček“, úhlových závorek a různých dodatečných kvalifikátorů dokáže kód značně znepřehlednit.
- Šablonový kód se velmi špatně ladí. Syntaktickou či sématickou chybu překladač u složitějších konstrukcí často hlásí chybovou zprávou o rozashu až několika řádků. Dekódovat takovou zprávu a přijít na to, kde je chyba, může být dosti nesnadné.
- Absence reflexe v době překaldu znemožňuje předat jako parametr šablony identifikátor a potenciálně brání některým užitečným technikám, například vygenerování serializátoru třídy v době překladu. V některých případech se toto omezení dá obejít, avšak mnohdy za cenu použití praktik vedoucích k ještě většímu zatemnění kódu.
- Složitý systém šablon rozprostřený přes několik hlavičkových souborů znamená dlouhé časy kompilace projektu.

3.5 Známá použití

Vysoká míra abstrakce a důraz na znovupoužitelnost předurčují šablony k využití v knihovnách. Implementátoři také této možnosti hojně využívají. Nyní si uvedeme několik nejznámějších a nejpoužívanějších.

3.5.1 STL

Standardní knihovna šablon (anglicky *standard template library*) nabízí širokou kolekci generických kontejnerů a algoritmů. Ve standardní knihovně mimo to najdeme také chytré ukazatele, šablonu *bind* pro adaptaci rozhraní funkce a svázání parametrů s hodnotami, n-tice (*tuples*) či rysy typů *type traits*.

3.5.2 Boost

Sada knihoven *boost* by se dala s trochou nadsázky označit za upstream pro standardní knihovnu jazyka C++. Je složena z mnoha součástí a velká část z nich hojně využívá šablonového metaprogramování.

Například knihovna *MPL* je přímo navržena jako podpora pro metastruktury metaalgoritmy. Nabízí obdobu knihovny STL, avšak zpracovanou v době překladu. Knihovna *Boost.Fusion* kombinuje metaprogramování a dynamické struktury. Jejím základem je šablona *tuple*.

Boost.Spirit umí vygenerovat syntaktický analyzátor (parser) pomocí syntaxe podobné BNF zapsené přímo jako inline C++ kód. *Boost.Signal* provádí volání funktorů s topologií one-to-many. *Boost.Python* zvládá vygenerovat rozhraní funkcí a tříd C++ pro jazyk Python pomocí jednoduché syntaxe zapsané přímo v jazyce C++.

Boost nabízí také nepřeberné množství různých pomocných knihoven, knihoven pro matematické výpočty, knihovnu pro práci s grafy a mnoho dalších...

3.5.3 Blitz++

Knihovna určená pro věděcké matematické výpočty. Obsahuje vektory, matice (zobecněné pro n rozměrů) a operace nad nimi. Díky použití šablon a přetěžování operátorů se práce s těmito pokročilými strukturami podobá jejich používání například v MATLABu. Výpočty jsou však mnohem efektivnější.

3.5.4 Loki

Knihnovna, kterou navrhl a popsal Andrei Alexandrescu v knize *Modern C++ Design* [2]. Obsahuje kolekci extrémně konfigurovatelných chytrých ukazatelů, šablonovou implementaci některých návrhových vzorů, generátory hierarchií tříd a podporu pro větvení do vláken.

Kapitola 4

Příklady

Dostáváme se k samotným ukázkám uplatnění šablonového metaprogramování v C++. Nebudeme se příliš zabývat klasickým použitím šablon ke tvorbě generických kontejnerů a tím duplikovat nejen standardní knihovnu, ale i téměř jakoukoli učebnici C++ [15]. Podíváme se rovnou na méně tradiční, ale o to zajímavější aplikace metaprogramování.

Struktura příkladů

Popisy jednotlivých příkladů se snaží dodržet jednotné schéma. Nejdříve si vytyčíme cíl, tedy nějaký problém, který chceme vyřešit. Protože se pohybujeme ve světě šablon, budeme se pokoušet najít řešení pokud možno co nejvíce obecné. Následuje samotný popis řešení, často doprovázený ukázkami zdrojového kódu v C++. Kvůli úrovni abstrakce takového kódu jej doplníme i o ukázku použití, poněvadž to nemusí být na první pohled zřejmé. V opodstatněných případech bude též uveden zástupce nějakého projektu z praxe, který danou techniku používá. Pokud bude použita nějaká ustálená technika, vzor či idiom často se vyskytující při tvorbě šablonových metaprogramů, bude tato skutečnost řádně zdůrazněna a okomentována. Většina takových zažitých postupů má svůj název. K dohledání konkrétního idiomu lze využít rejstříku.

Organizace příkladů

Zdrojové kódy k příkladům jsou uspořádány do modulů. Stěžejním je adresář `include` obsahující hlavičkové soubory s definicemi šablon. Jeho podadresáře odpovídají jednotlivým modulům a (velmi) přibližně kopírují rozdělení sekcí této kapitoly. Každý modul leží ve svém jmenném prostoru. Moduly jsou takovéto:

- **simple** – jednoduché demonstrační šablony bez valného využití v praxi
- **meta** – knihovna meta-struktur a meta-algoritmů
- **types** – šablony pro práci s datovými typy
- **patterns** – implementace některých návrhových vzorů
- **util** – užitečné šablony nespadající do jiné kategorie

Adresář `examples` obsahuje samotné příklady použití těchto šablon.

4.1 Jednoduché šablony pro začátek

Nyní se již vrhneme na psaní šablon. Začneme jednoduchými, avšak v praxi již použitelnými šablonami, na kterých si vyzkoušíme vybrané základní techniky.

4.1.1 Zábrana kopírování

C++ ke každé námi definované třídě automaticky vytvoří podporu pro kopírování, tj. kopírovací konstruktor a operátor přiřazení. Občas je nám tato skutečnost na obtíž, protože chceme naopak kopírování instancí naší třídy zabránit.

Ukázka 4.1: Šablona NonCopyable

```
1 template <class T>
2 class NonCopyable {
3 protected:
4     NonCopyable() {}
5     ~NonCopyable() {}
6 private:
7     NonCopyable(const NonCopyable&);
8     T& operator=(const T&);
9 };
```

Kopírovací konstruktor a operátor přiřazení jsou přesunuty do privátní sekce a tím zabráněno jejich použití. Bezparametrický konstruktor a destruktor jsou deklarovány jako protected, abychom zabránili přímému vytváření instancí tříd vzniklých z této šablony.

Ukázka 4.2: Použití šablony NonCopyable

```
1 class DoNotCopyThis : private NonCopyable<DoNotCopyThis> { /* ... */ };
```

CRTP Všimněme si zvláštního vzoru, kdy třída dědí od šablonové třídy, které předá jako parametr sebe sama. Tomuto idiomu se říká *curiously recurring template pattern*, zkráceně CRTP, a má při psaní šablon skutečně široké využití. Uplatní se například pro obohacení třídy o nové členské funkce, realizaci statického polymorfismu a mnoha při dalších příležitostech.

4.1.2 Čítač instancí

Nyní navrhneme třídu, která počítá své instance a tím i instance třídy, která ji má jako základní třídu, či členskou proměnnou.

Ukázka 4.3: Čítač instancí

```
1  template <class T>
2  class InstanceCounter {
3  public:
4      InstanceCounter()   { ++active_; ++total_; }
5      InstanceCounter(const InstanceCounter&) { ++active_; ++total_; }
6      ~InstanceCounter()  { --active_; }
7      static int active() { return active_; }
8      static int total()  { return total_; }
9  private: static int active_; static int total_;
10 };
11 template <class T> int InstanceCounter<T>::active_ = 0;
12 template <class T> int InstanceCounter<T>::total_  = 0;
```

Ukázka 4.4: Použití čítače instancí

```
class MyClass : private InstanceCounter<MyClass> { /* ... */ };
// somewhere in the code:
int current_myclass_instance_number = InstanceCounter<MyClass>::active();
```

Je zaznamenána informace o tom, kolik objektů bylo vytvořeno celkem a kolik jich existuje právě teď. V ukázce 4.5 vidíme již dobře známý CRTP. Není to však nutností, náš čítač by stejně dobře fungoval i jako členská proměnná. Taktéž není nutné dodržet, aby násobnost relace čítač – třída byla 1:1. Můžeme definovat čítač pro celou rodinu tříd (které nemají společnou základní třídu) tak, že čítačům všech takových tříd předáme stejný typový parametr. Pokud nemůžeme nebo nechceme jako typový parametr jednu z počítaných tříd, můžeme použít třídu stvořenou právě pro tento účel, čili *tag*.

Tag Tagem (značkou) nazveme takový koncept třídy či struktury, kterou používáme pouze jako identifikátor předaný šabloně jako parametr. Taková třída vůbec nemusí fyzicky existovat. Jedná se de-facto o statickou obdobu výčtu (*enum*). Je dobrým zvykem každou skupinu sémanticky příbuzných tagů sdružit do zvláštního jmenného prostoru.

Ukázka 4.5: Použití čítače instancí s tagem

```
struct CountedClasses; // tag
class MyFirstClass : private InstanceCounter<CountedClasses> { /* ... */ };
class MySecondClass : private InstanceCounter<CountedClasses> { /* ... */ };
// somewhere in the code:
int classes_created_so_far = InstanceCounter<CountedClasses>::total();
```

4.1.3 Generování příbuzných operátorů

Máme za úkol navrhnout šablonu, která do definice třídy doplní chybějící operátory. Tento problém má dva stupně. Pro třídy, u kterých lze pouze zjistit rovnost, doplníme kontrolu opačnou. U tříd, které lze úplně uspořádat, budeme požadovat existenci funkce `compare`, která bere jako parametr druhý objekt téže třídy a vrací hodnotu z množiny $\{-1, 0, +1\}$, pokud je tato instance menší, rovna, respektive větší než druhá. K řešení druhého problému se dá s úspěchem využít implementace prvního.

Ukázka 4.6: Operátory porovnání

```
template <class T>
class EqualityComparable {
    friend inline bool operator!=(const T& a, const T& b){ return !(b==a); }
};

template <class T>
class Comparable : private EqualityComparable<T> {
    typedef const T& TT; // const reference to T
    friend inline bool operator==(TT a, TT b) { return a.compare(b) == 0; }
    friend inline bool operator< (TT a, TT b) { return a.compare(b) <  0; }
    friend inline bool operator> (TT a, TT b) { return a.compare(b) >  0; }
    friend inline bool operator<=(TT a, TT b) { return a.compare(b) <= 0; }
    friend inline bool operator>=(TT a, TT b) { return a.compare(b) >= 0; }
};
```

Naši šablonu použijeme jako kteroukoli jinou založenou na idiomu CRTP. Musíme si pouze dávat pozor, aby naše třída poskytovala požadované operace.

Barton-Nackman trick V ukázce 4.6 byl použit idiom charakteristický definicí spřátelené funkce přímo uvnitř třídy zvaný *Barton-Nackman trick*. Takové funkce se berou v úvahu při dohledání symbolů na základě typů argumentů. Proto bude předchozí příklad po zpracování překladačem dodržujícím standard fungovat dle záměru.

4.1.4 Tisk na výstup

Ne každému vyhovuje syntaxe objektu `std::cout` pro tisk na výstup. Stvoříme si tedy generickou funkci, která bere proměnný počet parametrů a jejíž vlastnosti je možné v době překladu upravit.

Nejdříve se musíme rozhodnout, jakým způsobem bude naše funkce produkovat výstup. Vyvstávají otázky: Jak budou na výstupu odděleny jednotlivé parametry? Bude se tisknout do textového proudu (např. standardní výstup), na grafické zařízení či někam úplně jinam? Měl by se nějak označit začátek a konec výstupu jednoho volání funkce? V prvním odstavci jsme si vytyčili, že se bude jednat o funkci *generickou*. Teď dodáme, že to bylo myšleno i ve smyslu výše uvedených vlastností.

Půjde o funkci globální, což s sebou nese jistá omezení. Například že zařízení pro tisk musí být taktéž globálně dostupné (což `std::cout` splňuje). Také nebude mezi voláními uchovávat stav… a když, tak také globálně.

Jeden z cílů může být vyrobit funkci println, která vytiskne svoje argumenty hned po sobě (bez oddělovače) ukončené znakem nového řádku na standardní výstup.

Ukázka 4.7: Příklad použití println()

```
1 // prints "(3 + 4) / 2 = 3.5\n"
2 println("(3 + 4) / 2 = ", (3 + 4) * 0.5f);
```

Koncept tiskové třídy Navrhněme tedy koncept, který musí splňovat třída, která se bude starat o samotný tisk. Tato třída bude poté šablonovým parametrem pro naši generickou funkci.

Musí implementovat metodu *print*, která dostane jako parametr data k vytištění a musí být schopná zpracovat jakýkoli typ, který budeme chtít tisknout. Dále pak metody *before* a *after*, které budou zavolány před respektive po tištění samotných parametrů. Nakonec metoda *delimiter* bude mít na starosti vytištění oddělovače. Tato bude volána pouze v případě, že naše generická funkce dostane dva a více parametrů. Životnost objektu této třídy je omezena jedním voláním tisknoucí funkce.

Variadické šablony Povaha naší funkce pro tisk ji předurčuje k implementaci pomocí šablon s proměnným počtem typových parametrů. Operátor tři tečky má v tomto kontextu dva významy. V šablonovém parametru značí, že daný parametr zastupuje více argumentů. Uvnitř třídy či funkce má význam rozbalení parametru na daném místě.

Ukázka 4.8: Variadické šablony

```
1 template <typename... Types>
2 inline void func(Types... args) { do_something(x(args)...); }
3
4 func(3, "one", 2.2); // calls: do_something(x(3), x("one"), x(2.2));
```

Funkce `generic_print` tedy bude muset vytvořit instanci tiskové třídy a následně ji použít k vytištění uvození výstupu, vytištění parametrů a ukončení výstupu.

Ukázka 4.9: Funkce generic_print

```
template <class Printer, class... Types>
inline void generic_print(const Types&... members) {
    Printer p;
    p.before();
    generic_print_impl(p, members...);
    p.after();
}
```

Samotné vytištění parametrů bude mít několik možných verzí, proto jej realizujeme zvláštní sadou přetížených funkcí. První funkce z ukázky 4.10 obsluhuje případ, kdy nedostaneme ani jeden parametr, druhá pak když dostaneme parametr právě jeden.

Třetí má pak na starosti všechny ostatní případy, bude tedy tisknout i oddělovač. Nejdříve vytiskne první parametr ze seznamu, poté oddělovač a nakonec zavolá zase naši přetíženou funkci pro zbytek argumentů. Přestože funkce jakoby volá „sama sebe“, nejde o rekurzi. Volaná funkce totiž představuje jinou instanci šablony. Jelikož je uvedeno klíčové slovo `inline`, s velkou pravděpodobností nepůjde ani v pravém slova smyslu o volání funkce. Generování funkcí se zastaví voláním instance druhé šablony, až zůstane jediný parametr.

Ukázka 4.10: Funkce generic_print_impl

```
template <class Printer>
inline void generic_print_impl(Printer&) { }

template <class Printer, class Type>
inline void generic_print_impl(Printer& p, const Type& x) { p.print(x); }

template <class Printer, class FirstType, class... RestTypes>
inline void generic_print_impl(Printer& p,
            const FirstType& x, const RestTypes&... rest) {
    p.print(x);
    p.delimiter();
    generic_print_impl(p, rest...);
}
```

Nyní již je téměř dosaženo cíle, funkce `println`. Nejdříve však napíšeme šablonovou třídu, která bere jako parametr referenci na standardní výstupní proud, kam bude směřovat její výstup. Tisk ukončuje odřádkováním a neprodukuje žádné oddělovače.

Ukázka 4.11: Tisková třída a funkce println

```
template <std::ostream& out>
struct StreamPrinter {
    template <typename T>
    void print(const T& x) { out << x; }
    void before()    {  }
    void delimiter() {  }
    void after()     { out << std::endl; }
};

template <class... Types> void println(Types... args) {
    generic_print<StreamPrinter<std::cout>>(args...);
}
```

V přiloženém balíčku zdrojových kódů s příklady naleznete také velmi podobným stylem (t.j. tisková třída a „obalovací“ šablona) realizovanou funkci `dump`. Její výstup je o poznání bohatší – řádně označí začátek a konec výstupu, jednotlivé parametry očísluje a u některých napíše i jejich datový typ.

Naše generická výstupní funkce tedy poskytuje algoritmus tištění: nějakým způsobem uvozený a ukončený výstup několika oddělených objektů. Vytváří abstrakci nad výstupním zařízením i nad reprezentací dat na výstupu. Odstiňuje tvůrce nového způsobu tisku objektů od práce s variadickými šablonami.

Pozoruhodná je taktéž podobnost s návrhem ve světě objektově orientovaných technologií. Naše šablonová funkce de-facto implementuje statickou obdobu návrhového vzoru *template method*[1].

4.2 Meta-struktury a meta-algoritmy

Posuneme se trochu dále. Podíváme se na základní struktury a s nimi související algoritmy, které pracují čistě v době překladu.

4.2.1 Primitivní hodnota

Parametrem šablony může být i primitivní hodnota, například typu `int`. Vzhledem k tomu, že se ale metaprogramování pomocí šablon točí převážně kolem datových typů, bude se nám hodit nástroj, který „namapuje“ skalární hodnoty na typy.

Ukažme si tedy výslednou šablonu. Jejími instancemi jsou přirozeně datové typy, čímž jsme dosáhli kýženého cíle. Jelikož nemá datový typ vzniklý pomocí této šablony za běhu programu žádný význam, bude volání konstruktoru zakázáno.

Ukázka 4.12: Šablona Value (meta-hodnota)

```
template <typename valtype, valtype val>
struct Value {
    typedef Value type;
    typedef valtype value_type;
    static const valtype value = val;
private: Value();
};
// use example: Value<int, 5>
```

Taktéž nadefinujeme pár pomocných šablon, které nám usnadní práci s takovýmito hodnotami. Pro booleovskou hodnotu si také vytvoříme pomocné šablony `True`, `False`.

Ukázka 4.13: Pomocné šablony pro primitivní typy

```
template <char     x> struct Char  : public Value<char,     x> {};
template <int      x> struct Int   : public Value<int,      x> {};
template <unsigned x> struct UInt  : public Value<unsigned, x> {};
template <size_t   x> struct SizeT : public Value<size_t,   x> {};
template <bool     x> struct Bool  : public Value<bool,     x> {};
// "metaconstants"
typedef Bool<true>  True;
typedef Bool<false> False;
```

Z kteréhokoli z takto získaných typů můžeme získat „základní verzi“ hodnoty následovně: `Int<5>::type`. Výsledkem takového výrazu je typ `Value<int, 5>`.

[1] V OOP nemá slovo template použité v tomto názvu co do činění s metaprogramováním.

Operace nad meta-hodnotami Nad mnoha meta-hodnotami mají smysl běžné operace, jako například sčítání. Šablonu, která provádí transformaci datových typů (v našem případě zatím jen meta-hodnot) nazveme *metafunkcí*. Návratová hodnota metafunkce (která je ve skutečnosti datovým typem) se vrací v členu šablony nazvaném `type`. Pokud chceme volání delegovat na další metafunkci, můžeme s úspěchem využít dědičnosti.

Ukázka 4.14: Součet meta-hodnot

```
1  template <class a, class b>
2  struct Add : public Add<typename a::type, typename b::type> {};
3
4  template <class T, T a, T b>
5  struct Add<Value<T, a>, Value<T, b>> : public Value<T, (a + b)> {};
```

Nespecializovaná verze pouze „vybalí“ argumenty z případných pomocných šablon. Specializovaná provádí samotné sčítání. Vidíme, že je podporována pouze přesná shoda typů. Specializace sčítání pro další datové typy se dají dopsat i dodatečně. Musí však ležet v témže jmenném prostoru.

4.2.2 Seznam

Dále vytvoříme šablonu pro sekvenci datových typů. Definice v C++0x je přímočará.

Ukázka 4.15: Meta-seznam

```
1  template <class... items>
2  struct List {
3      typedef List type;
4      typedef SizeT<sizeof...(items)> size;
5      static const size_t length = size::value;
6  private: List();
7  };
```

Vidíme využití meta-hodnoty typu `SizeT` k vyjádření délky seznamu.

Seznam hodnot Náš seznam dokáže pomocí výše popsaných meta-hodnot pojmout i primitivní typy. Zápis takového seznamu je však mírně řečeno nepohodlný. Taktéž by se hodil jednoduchý způsob, kterak dostat z takového seznamu hodnoty jako statické pole. Pomocí další šablony však dokážeme oba tyto problémy elegantně odstranit.

Ukázka 4.16: Meta-seznam hodnot stejného typu

```
1  template <typename valtype, valtype... vals>
2  struct ValueList : public List<Value<valtype, vals>...> {
3      static const valtype values[sizeof...(vals)];
4  };
5  template <typename valtype, valtype... vals>
6  const valtype ValueList<valtype, vals...>::values[] = { vals... };
7
8  template <typename itemtype, itemtype... listitems>
9  struct ValueList<List<Value<itemtype, listitems>...>> :
10         public ValueList<itemtype, listitems...> {};
```

Specializace vyobrazená na posledních třech řádcích ukázky 4.16 slouží k získání seznamu hodnot z obyčejného seznamu. Aby se nám pracovalo ještě lépe, nadefinujeme ještě šablonu `Range`, která nám nageneruje seznam hodnot ve zvoleném rozsahu. Ukázka je k mání v přiložených kódech.

Získání prvku Základní operací se seznamem je získání n-tého prvku. Provedeme to rekurzivně. Pro získání prvku s indexem nula rovnou vrátíme první prvek, získání jiného prvku znamená získání prvku o indexu $n-1$ ze seznamu bez prvního prvku.

Ukázka 4.17: Získání prvku seznamu

```
template <class list, class n>
struct Get : public Get<typename list::type, typename n::type> {};

template <class first, class... rest, int n>
struct Get<List<first, rest...>, Value<int, n>> {
    typedef typename Get<List<rest...>, Value<int, n-1>>::type type;
};

template <class first, class... rest>
struct Get<List<first, rest...>, Value<int, 0>> {
    typedef first type;
};

template <class number>
struct Get<List<>, number> {
    static_assert(number::value && !number::value, "Index out of range");
};

template <class list, int n>
struct GetN : public Get<typename list::type, Value<int, n>> {};
```

První nespecializovaná verze šablony zase pouze získá „surový“ seznam z případných pomocných šablon (seznam meta-hodnot či jejich rozsah). Tento vzor se bude často opakovat, takže v dalších případech budeme uvádět pouze příslušné specializace. Šablona `GetN` slouží pouze k usnadnění práce. Bere namísto meta-hodnoty jako parametr přímo index jako celé číslo.

Všimněte si konstrukce `static_assert`. Jedná se o další z nových vlastností C++0x. Bere jako vstup podmínku a hlášení, které překladač zobrazí jako chybu a odmítne přeložit celý program, pokud podmínka není splněna. Důležité je si uvědomit, že podmínkový výraz musí být závislý na šablonovém parametru, jinak bude vyhodnocen a celý překlad potenciálně selže i když nevznikne žádná instance dané šablony!

Spojení dvou seznamů Další základní operací je spojení více seznamů do jednoho. I to se dá definovat rekurzivně. Nejdříve musíme umět spojit seznamy dva. Konkatenaci více seznamů poté provedeme jako seznam vzniklý konkatenací prvního seznamu a seznamu vzniklého konkatenací ostatních seznamů.

Ukázka 4.18: Spojení seznamů

```
template <class... lists>
struct Join : public Join<typename lists::type...> {};

template <class... items, class... lists>
struct Join<List<items...>, lists...> :
    public Join<List<items...>, typename Join<lists...>::type> {};

template <class... items1, class... items2>
struct Join<List<items1...>, List<items2...>> :
    public List<items1..., items2...> {};

template <> struct Join<> : public List<> {};
```

Další operace V přiložených kódech je nad seznamy definováno ještě množství dalších operací. Nebudou zde podrobně rozebrány, ale některé další ukázky kódů s nimi mohou počítat, takže alespoň výčtovitě vzpomeneme některé z nich.

Jde o ověření přítomnosti prvku (`Contains`), zjištění pozice prvku (`IndexOf`), odebrání prvku (`Remove`), otočení seznamu (`Reverse`), transpozici vnořených seznamů (`Transpose`), prostrkaní dvou seznamů (`Zip`). Ve spojení s konceptem třídy metafunkce (viz níže) máme nadefinovány i metafunkce vracející nejmenší prvek seznamu podle relace zadané metafunkcí předané jako parametr (`MinElement`) a metafunkci k seřazení seznamu (`Sort`), taktéž parametrizovatelnou relační metafunkcí.

4.2.3 Podmínky

Podmínky jsou více či méně zjevnou součástí každého programovacího jazyka. V šablonách zastupuje jejich funkci specializace. Můžeme si však celou věc usnadnit a vyhnout se explozi syntaxe jednoduchou šablonou.

Switch Šablona `Switch` příjmá lichý počet parametrů, vždy výraz vracející booleovskou meta-hodnotu (c_i) a příslušnou návratovou hodnotu (r_i). Posledním parametrem je návratová hodnota pro případ, že nebude splněna ani jedna z podmínek (r_e). Syntaxe:

$$\text{Switch}(c_1, r_1, c_2, r_2, \ldots, c_n, r_n, r_e)$$

Výsledná šablona vrátí první hodnotu (r_1), pokud je první podmínka pravdivá (c_1), jinak vrací výsledek, který vzejde ze zpracování zbytku argumentů. Jakmile zbývá jediný parametr, jedná se o „else větev“ (r_e) a je tudíž návratovou hodnotou.

Ukázka 4.19: Šablona Switch

```
1  template <class iftrue, class... rest>
2  struct Switch<Value<bool, true>, iftrue, rest...> {
3      typedef iftrue type;
4  };
5
6  template <class iftrue, class nextcond, class... rest>
7  struct Switch<Value<bool, false>, iftrue, nextcond, rest...> :
8      public Switch<nextcond, rest...> {};
9
10 template <class elseval>
11 struct Switch<elseval> { typedef elseval type; };
```

If Pomocný adaptér rozhraní předchozí šablony přijímající vždy tři parametry, přičemž první je booleovská hodnota (ne meta-hodnota).

EnableIf Velmi užitečná šablonka, která nám umožní kontrolovat instanciaci šablon. Šablona bude instanciována pouze v případě splnění určité podmínky. Pro více inspirace viz zdrojové kódy nebo knihovny Boost.

Pod zkratkou SFINAE se skrývá pojem *Substitution Failure Is Not An Error* a také vysvětlení, proč `EnableIf` funguje. Umožňuje tak omezenou kontrolu konceptu šablonového parametru překladačem. Šablona `EnableIf` je totiž napsána tak, aby její instanciace selhala pro případ, že její první parametr nabude hodnoty *false*. To však překladač nepovažuje za chybu a vyzkouší jinou variantu dané šablony.

4.2.4 Metaprogramování vyššího řádu

Posuneme se ještě o úroveň výše. Doposud jsme používali metafunkce pouze k transformaci datových typů. V této části práce si ukážeme, jak napsat metafunkce, které přijímají jako parametr nebo vrací jiné metafunkce.

Parametrem šablony i členským atributem výsledné třídy může být šablona. Avšak vzhledem k tomu, že většina obecných meta-algoritmů a meta-struktur (například `List`) očekává jako parametry datové typy, přijde vhod podobná abstrakce, jakou jsme zavedli u primitivních hodnot [1].

Třída metafunkce Třída (či struktura), která obsahuje metafunkci `apply`, se nazývá *třída metafunkce*. Můžeme ji získat přímo z metafunkce jednoduchým „obalením“ do třídy.

Abychom to nemuseli provádět ručně, napíšeme si pro tento účel šablonu. Jejím parametrem bude šablona, kterou chceme přetvořit na třídu metafunkce, výslednou metafunkční třídu pak reprezentuje přímo instance této šablony.

Ukázka 4.20: Šablona Func pro vytvoření třídy metafunkce

```
template <template <typename...> class tmpl>
struct Func {
    template <typename... params>
    struct apply : tmpl<params...> {};
};
```

Poznámka: Pokud bude v následujícím výkladu použito slovo metafunkce a na příslušném místě bude parametr očekávající datový typ, budeme tím myšlena *třída* metafunkce.

Volání metafunkce z třídy je však značně nepohodlné, zvláště pak z šablonového prostředí, kde musíme členy dodatečně kvalifikovat.

Ukázka 4.21: Nepříjemné volání metafunkce z šablony

```
typedef typename SomeMetaFuncClass::template apply<par1, par2>::type result;
```

Těmto nepříjemnostem se vyhneme nadefinováním další pomocné šablony. Parametrem bude metafunkční třída a seznam parametrů, se kterými ji má zavolat. Její síla se skutečně ukáže až při psaní dalších metafunkcí, které volají metafunkce jim předané jako parametr. Taktéž poznamenejme, že tato šablona je sama o sobě metafunkcí vyššího řádu.

Ukázka 4.22: Šablona pro volání třídy metafunkce

```
template <class metafunc, typename... items>
struct MetaCall<metafunc, List<items...>> :
    public metafunc::template apply<items...> {};
// MetaCall<Func<Add>, ValueList<int, 3, 5>>::type ==> Value<int, 8>
```

4.2.5 Algoritmy

Nyní si ukážeme některé generické meta-algoritmy. Ve značné míře využijí výše uvedených technik. Často operují nad seznamy, mohou dokonce i zpracovat seznam metafunkcí.

Ukázka 4.23: Ukázka použití meta-algoritmů vyššího řádu

```
typedef List<Func<Add>, Func<Sub>, Func<Mul>> MetaFuncs; // funcs to call
typedef BindLast<Func<MetaCall>, ValueList<int, 12, 3>> Binder; // arguments
typedef ValueList<ForEach<Binder, MetaFuncs>::type> Result; //call each func
const int* values = Result::values; // contains { 15, 9, 36 }
```

ForEach Výsledkem šablony `ForEach` je seznam prvků po aplikaci metafunkce na každý prvek původního seznamu. Implementace pomocí variadických šablon je přímočará, neboť není třeba iteraci simulovat rekurzí.

Ukázka 4.24: Šablona ForEach

```
template <class metafunc, class... items>
struct ForEach<metafunc, List<items...>> :
    public List<typename MetaCall<metafunc, List<items>>::type...> {};
```

Accumulate Šablona aplikuje binární funkci f od dané startovní hodnoty a_0 postupně přes všechny prvky seznamu x_i o délce n. Lze ji definovat též rekurzivně, což se nám bude hodit při implementaci šablonou. Potom je však seznam potřeba nejdříve převrátit.

$$\text{Accumulate}(f, a_0, \mathbf{x}) = f(f(\ldots f(f(a_0, x_1), x_2)\ldots, x_{n-1}), x_n)$$

Ukázka 4.25: Šablona Accumulate

```
template <class metafunc, class initval, class first, class... rest>
struct Accumulate<metafunc, initval, List<first, rest...>> :
    public MetaCall<metafunc,
        List<Accumulate<metafunc, initval, List<rest...>>, first>> {};

template <class metafunc, class initval>
struct Accumulate<metafunc, initval, List<>> { typedef initval type; };
```

Bind Tato šablona je bez bližšího vysvětlení použita v ukázce 4.23. Metafunkce `BindLeft` (respektive `BindRight`) příjmá jako argument metafunkci a jejích prvních (respektive posledních) n parametrů nastaví napevno podle zbytku argumentů. Výslednou metafunkci vrátí. V uvedeném příkladě nastaví metafunkci `MetaCall` tak, aby metafunkci předanou jí jako parametr volala vždy s pevně určenými parametry reprezentujícími hodnoty 12 a 3.

4.2.6 Faktoriál ještě jednou

Nyní spojíme předešlé meta-struktury a meta-algoritmy. Vrátíme se k příkladu faktoriálu, avšak tentokrát jej implementujeme poněkud sofistikovaněji jako metafunkci. Nakonec jej použijeme k vygenerování statického pole, do kterého si za překladu předpočítáme hodnoty několika prvních faktoriálů. Tyto pak můžeme využít a z funkce určené pro běh programu je rovnou vrátit, aniž by musely být počítány.

Pro výpočet samotného faktoriálu použijeme šablonu `Accumulate` s operací násobení přes rozsah hodnot. Speciální případ pro 0! musíme ošetřit zvláštní specializací. Pole s faktoriály pak vygenerujeme z rostoucího rozsahu meta-algoritmem `ForEach`.

Ukázka 4.26: Faktoriál pomocí meta-algoritmů

```
template <class x> struct Factorial :
    Accumulate<Func<Mul>, UInt<1>, ValueRange<unsigned, 1, x::value>>::type
    { static_assert(x::value > 0, "Factorial from a negative number!"); };
template <typename inttype> struct Factorial<Value<inttype, 0>> :
    public UInt<1>::type {};

typedef ValueList<ForEach<Func<Factorial>,
        ValueRange<unsigned, 0, 12>>::type> FACTORIALS;
```

4.2.7 Práce s datovými typy

Práce se šablonami se točí převážně okolo datových typů. Často přijde vhod mít o daném datovém typu informace navíc. Například zda je efektivnější předat jej funkci odkazem nebo hodnotou, zda se jedná o ukazatel, rozsah celočíselného typu, počet a typy parametrů ukazatele na funkci a mnoho dalších údajů. Těmto údajům říkáme rysy typu (*type traits*).

Klasifikátory

Šablony mohou poskytovat informace o datovém typu. Ukažme si to na příkladu rozpoznání ukazatele od jiného datového typu. Pozitivní případ bude reprezentován meta-hodnotou `True`, negativní jako `False`. Základní šablona bude odpovídat základním typům, specializovaná verze ukazatelům.

Ukázka 4.27: Detekce ukazatele

```
1 template <typename t> struct IsPtr     : public False {};
2 template <typename t> struct IsPtr<t*> : public True  {};
```

Manipulátory

Šablony mohou podobným způsobem typy modifikovat. Když už jsme u ukazatelů, mohou například ukazatele přidávat či naopak likvidovat. Podobně je možné přidávat či odstraňovat reference, kvalifikátory `const` a `volatile`, zjišťovat, zda je možno třídu kopírovat či vytvořit konstruktorem bez parametrů a tak dále.

Ukázka 4.28: Odstranění jednoho ukazatele

```
1 template <typename t>
2 struct RemovePtr { typedef t type; };
3 template <typename t>
4 struct RemovePtr<t*> { typedef t type; };
5
6 typedef ForEach<Func<RemovePtr>, List<int***, char*, string>>::type MyTypes;
7 // MyTypes is List<int**, char, string>
```

Funkční typy

Funkční typ v C++ popisuje rozhraní funkce. Jde o něco trochu jiného než typ ukazatel na funkci. Vlastně můžeme funkční typ získat z typu ukazatel na funkci pomocí výše vyobrazené šablony `RemovePtr`. Z funkčního typu můžeme vyčíst počet a typy parametrů funkce a také návratovou hodnotu.

To můžeme využít například pro generování rozhraní pro meziprocesovou komunikaci. Systém D-Bus posílá v každé zprávě signaturu volané metody. S pomocí této šablony ji můžeme přímo extrahovat a vygenerovat.

Ukázka 4.29: Rysy pro funkční typy

```
1 template <class rettype, class... params>
2 struct Function<rettype(params...)> {
3     typedef List<params...> args;
4     static const int arity = args::length;
5     typedef rettype return_type;
6     template <int n> struct arg : public GetN<args, n> {};
7 };
```

4.2.8 Řetězce

Podporu pro práci s textovými řetězci v době překladu již v podstatě máme. Stačí `ValueList` rozšířit tak, aby z něj bylo možno získat statické pole ukončené nulovým znakem.

Nevýhodou bude samotný zápis řetězce. Parametrem šablony sice může být ukazatel, ale k datům reprezentovaným tímto ukazatelem překladač během zpracovávání šablon přistoupit nemůže. Budeme tedy muset jednotlivé znaky řetězce odděleně vyjmenovat jako parametry šablony. Pro zjednodušení budeme uvažovat pouze řetězce se znaky typu `char`.

Ukázka 4.30: Šablona pro znakové meta-řetězce

```
template <char... chrs>
struct String : ValueList<char, chrs...> {
    typedef ValueList<char, chrs...> vallist;
    static const size_t strlen = vallist::length + 1;
    static const char str[strlen];
};

template <char... chrs>
const char String<chrs...>::str[] = { chrs..., '\0' };
```

Protože řetězec splňuje koncept seznamu, vlastně už jej umíme spojovat a různě modifikovat. Potřebujeme už pouze nástroj, který nám z výsledného seznamu vyrobí zase řetězec.

Ukázka 4.31: Získání řetězce ze seznamu znaků

```
template <char... chrs>
struct StringFromList<List<Value<char, chrs>...>> {
    typedef String<chrs...> type;
};
```

Nyní se podíváme na některé možnosti využití řetězců zpracovaných za překladu.

Syntéza jmen typů Nyní máme možnost vytvořit šablonu `TypeToString`, převede typ na jeho řetězcovou reprezentaci v nějakém formátu. To je užitečné například u mnohých komunikačních protokolů, které v odesílaných zprávách často uvádějí i informaci o typu zapouzdřených dat.

Neznámé typy odchytí nespecializovaná verze šablony a lze je ošetřit buď pomocí nějaké výchozí hodnoty nebo ukončit překlad užitím statické aserce. Každý typ bude mít svou specializaci této metafunkce, některé jsou uvedeny v ukázce 4.32.

Ukázka 4.32: Konverze datového typu na jeho řetězcovou representaci

```
template <> struct TypeToString<int> { typedef String<'i','n','t'> type; };
template <> struct TypeToString<bool> {
    typedef String<'b','o','o','l'> type; };
template <typename t> struct TypeToString<t*> :
    public StringFromList<Join<TypeToString<t>, String<'*'>>> {};
```

Binární literály Tyto řetězce je možno také použít k výpočtu hodnoty binárního či jiného literálu za překladu (ačkoli příklad v přiložených kódech tak nečiní).

V tomto případě se dokonce můžeme vyhnout vkládání literálu znak po znaku do šablony, máme-li k dispozici podporu pro uživatelsky definované literály. Ty však zatím překladače neimplementují.

4.3 Meta-algoritmy a dynamický kód

Jak bylo předvedeno v předchozí sekci, šablony představují ve své doméně plnohodnotný programovací jazyk. Byla by však škoda síly výše uvedených postupů využít pouze k vygenerování několika konstant. Podívejme se tedy, jak nám pomohou při tvorbě kódu zpracovávaného samotnou výslednou aplikací.

4.3.1 Hierarchie tříd

Ze seznamů můžeme generovat celé hierarchie tříd, jak je popsáno v [2]. My však máme k dispozici navíc koncept metafunkce a variadické šablony.

Izolovaná hierarchie Izolovaná hierarchie představuje případ, kdy třída dědí od několika dalších nezávisle na sobě s užitím vícenásobné dědičnosti. Nejdříve navrhneme šablonu, která očekává jako parametr seznam a dědí od každého prvku daného seznamu.

Ukázka 4.33: Generátor izolované hierarchie

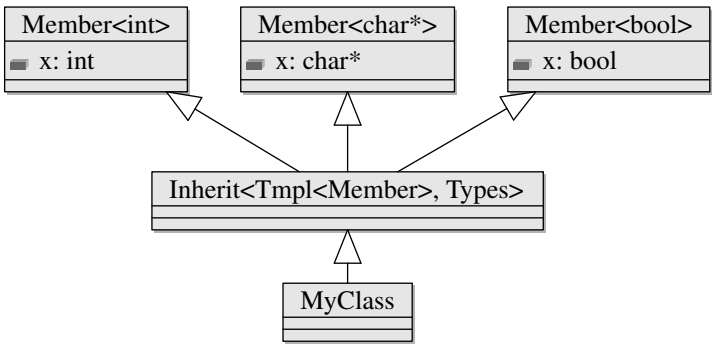
```
1 template <class... types>
2 struct Inherit<List<types...>> : public types... {};
```

Často se nám však hodí nedědit přímo od typů v seznamu, ale od typů z nich vzniklých nějakou šablonou. V ten moment může nastoupit algoritmus `ForEach`. Nejdříve však potřebujeme šablonu `Tmpl`, která podobně jako `Func` vyrobí z šablony třídu metafunkce, avšak s tím rozdílem, že tentokrát je návratová hodnota zapouzdřené metafunkce přímo instance dané šablony s příslušnými parametry.

Ukázka 4.34: Příklad generování izolované hierarchie tříd

```
1 template <typename T> class AddMember { public: T x; };
2 typedef List<int, char*, bool> Types;
3 class MyClass : public Inherit<ForEach<Tmpl<AddMember>, Types>::type> {};
```

Vygenerovaná hierarchie vypadá následovně:



Lineární hierarchie Lineární hierarchie používá dědičnost jednoduchou. Navíc je potřeba jistá spolupráce generující šablony. Druhý parametr, který přijme, musí použít jako svého rodiče. První již udává přímo parametrizaci této instance v hierarchii. Šablona generující jednotku v hierarchii bude tentokrát přímo parametrem generátoru.

Pro zdrojový text generátoru lineárních hierarchií `LinearInherit` nahlédněte do přiložených kódů. Parametry představují v tomto pořadí generující metafunkci, seznam parametrů jednotek v hierarchii a třída představující samotný vršek hierarchie.

4.3.2 N-tice

Při definování seznamu typů jsme byli jen krůček od popisu n-tice (*tuple*). Ta má navíc ke každému typu přiřazenou hodnotu [8]. Tato heterogenní datová struktura poskytuje mnohem širší využití v programech za jejich běhu.

Obsahuje operace pro získání prvního prvku n-tice, pro získání n-tice zbytku prvků bez prvního, nečlenskou šablonu `get<N>(tuple)` pro získání N-tého prvku.

Ukázka 4.35: Šablona Tuple

```
template <typename first_t, typename... rest_t>
class Tuple<first_t, rest_t...> : private Tuple<rest_t...> {
public:
    typedef first_t MemberType;                    // member type
    typedef Tuple<rest_t...> RestType;             // parent type
    typedef meta::List<first_t, rest_t...> Types; // type list
    static const size_t size = Types::length;      // tuple size

    Tuple() {}

    Tuple(const MemberType& mem, const rest_t&... rest) :
        RestType(rest...), member(mem) {}

    template <typename... other_t>
    Tuple(const Tuple<other_t...>& rhs) :
        RestType(rhs.rest()), member(rhs.first()) {}

    template <typename... other_t>
    Tuple& operator=(const Tuple<other_t...>& rhs) {
        member = rhs.first(); rest() = rhs.rest(); return *this;
    }

    MemberType& first() { return member; }
    const MemberType& first() const { return member; }

    RestType& rest() { return static_cast<RestType&>(*this); }
    const RestType& rest() const {
        return static_cast<const RestType&>(*this);
    }

    template <int N> struct TypeOf : public meta::GetN<Types, N> {};

private:
    MemberType member;
};
```

Pro snadnější vytváření instancí navrhneme funkci `mktuple`, která spolehlivě odvodí typ n-tice na základě typů argumentů. Dále funkce `tie` vytváří podobným způsobem n-tici referencí. Například prohození hodnot proměnných pak lze realizovat velmi jednoduše.

Ukázka 4.36: Prohození obsahů proměnných pomocí n-tice

```
tie(a, b) = mktuple(b, a);
```

N-tice nám vytvoří základ pro elegantní spojení světa meta-algoritmů a dynamického prostředí běhu programu. Velká část technik manipulujících se seznamy z předchozí sekce se dá s trochou představivosti aplikovat také na n-tici.

Volání Jednou z užitečných operací se ukáže být volání funkce s prvky n-tice jakožto parametry. Napíšeme si šablonu `call`, která to umožní. Konkrétní implementace je k nalezení v přiložených kódech, zde si pouze nastíníme možné způsoby řešení.

Můžeme postupně vybalovat jednotlivé prvky a předávat je jako parametry naší šablony, vždy n-tici se zbytkem parametrů, dále již vybalené prvky a první prvek n-tice. Specializovaná verze pro prázdný tuple jen zavolá danou funkci. Postup volání bychom mohli vyobrazit následovně (hranaté závorky představují n-tici, f volanou funkci):

$$\text{call}(f, [p_1, p_2]) \to \text{call}(f, [p_2], p_1) \to \text{call}(f, [], p_1, p_2) \to f(p_1, p_2)$$

Druhou možností je vytvořit rozsah celočíselných hodnot $(0, 1, 2, \ldots, size-1)$ představující indexy jednotlivých prvků. Když máme toto k dispozici, volání již provedeme jednoduše (ilustrační kód):

Ukázka 4.37: Volání funkce pomocí n-tice a rozsahu indexů

```
template <class FuncType, class TupleType, unsigned... indexes>
typename return_of<FuncType>::type call_impl(FuncType f, TupleType t) {
    return f(get<indexes>(t)...);
}
```

Tuto šablonu využijeme, známe-li parametry funkce, avšak nechceme ji hned volat, tak si parametry uložíme do n-tice. To může nastat například když ještě nevíme, která funkce bude volána či ji chceme volat se zpožděním. Například funkce `bind`, která se dostala do standardní knihovny, používá podobnou techniku a formální parametry ukládá do n-tice. Jiné využití bude představeno v sekci 4.4.2 o návrhovém vzoru *proxy*.

ForEach Další funkce také sdružuje n-tice s funkcemi, avšak trochu jiným způsobem. Parametrem zvolená funkce či funkční třída je volána pro každý člen n-tice. Pokud je parametrem obyčejná funkce, musí být tato typově kompatibilní se všemi prvky n-tice.

Funkční třída poskytuje mnohem širší možnosti využití. Operátor volání funkce totiž může být přetížen pro více datových typů či může jít o šablonu a pro každý prvek tedy může být efektivně volána jiná implementace. Tím jsme dosáhli statického polymorfního chování. Navíc každá verze přetíženého operátoru může být zvlášť dodatečně podle potřeba kvalifikována jako `inline`.

Ukázka 4.38: Funkce ForEach pro n-tice

```
template <typename Functor>
inline void foreach_impl(Functor, Tuple<>) {}

template <typename Functor, typename First, typename... Types>
inline void foreach_impl(Functor f, Tuple<First, Types...>& tuple) {
    f(tuple.first());
    foreach_impl<Functor, Types...>(f, tuple.rest());
}

template <typename Functor, typename... Types>
inline void foreach(Functor f, Tuple<Types...>& tuple)
    { foreach_impl<Functor, Types...>(f, tuple); }
```

Stojí za zmínku, že výše uvedený kód má stejný výsledek, jako bychom danou funkci volali v několika příkazech pod sebou na každý člen n-tice, což je následkem použití `inline`. Nejde ani o rekurzi, každé volání foreach vytvoří novou instanci.

4.4 Návrhové vzory

Návrhové vzory pro objektově orientované programování někteří považují za chybějící vlastnosti programovacího jazyka. Pokud však dokážeme vzor zapracovat do knihovny, dokážeme, že jazyk podporu pro danou vlastnost poskytuje, ač nepřímo.

Implementace následujících návrhových vzorů více či méně využívají technik a nástrojů uvedených dříve.

4.4.1 Prototyp

Realizace prototypu, jak si jej představíme zde, spočívá v přidání virtuální členské funkce `clone`. Ta bude mít za úkol vrátit ukazatel na kopii daného objektu bez ohledu na to, zda máme k dispozici ukazatel či referenci přímo na daný objekt či na jeho předka.

Vytvoříme tedy základní třídu, od které může dědit třída na samotném vršku klonovatelné hierarchie. Poskytuje abstraktní funkci `clone` a virtuální destruktor. Není nutné ji použít, požadované funkce je stejně tak dobře možné do předka všech klonovatelných tříd napsat ručně.

Ukázka 4.39: Bázová třída pro hierarchii klonovatelných tříd

```
template <class Base> struct Cloneable {
    virtual Base* clone() const = 0;
    virtual ~Cloneable() {}
};
```

Nyní bude naším cílem vytvořit šablonu, která se stará o samotné kopírování objektů. Využije dobře známého vzoru CRTP, který ještě navíc podpoříme makrem `SELF`. To vrací náš objekt, avšak staticky přetypovaný na parametrem určeného potomka. První parametr šablony při dědění ve třídě tedy bude třída samotná, druhý bude její „plnohodnotný“ předek v hierarchii. Naše šablona implementující klonování se tedy vloží mezi ně.

Samotná duplikace bude řízena kopírovacím konstruktorem.

Ukázka 4.40: Klonovač

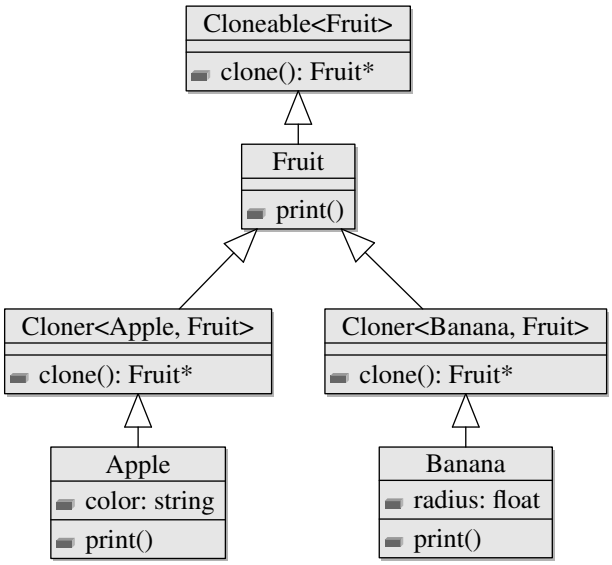
```
template <class Deriv, class Base>
struct Cloner : public Base {
    Base* clone() const { return new Deriv(*SELF(const Deriv)); }
};
```

Nyní si ukážeme, jak umožnit instancím ovoce klonování metodou `clone`.

Ukázka 4.41: Použití klonovače

```
// base class (cloneable fruit)
struct Fruit : public Cloneable<Fruit> {
    virtual void print() const = 0;
};
struct Banana : public Cloner<Banana, Fruit> {
    float radius;
    explicit Banana(float r) : radius(r) {}
    void print() const;
};
struct Apple : public Cloner<Apple, Fruit> {
    string color;
    explicit Apple(const string& clr) : color(clr) {}
    void print() const;
};
```

Naklonovaná jablka zachovávají barvu. Stejně tak klony banánů mají stejný poloměr zakřivení jako jejich předloha. Hierarchii tříd vzniklou z ukázky 4.41 znázorňuje následující diagram.



4.4.2 Proxy

Vzor proxy má zprostředkovávat rozhraní k jinému objektu. Může jít o síťové spojení, další objekty v paměti či zprostředkování komunikace mezi procesy.

Známým příkladem jsou chytré ukazatele, které se navenek chovají jako ukazatel na daný objekt, ale přidávají funkčnost správy zdrojů, například počítání referencí.

Odložená konstrukce objektu Představme si situaci, kdy máme objekt, u kterého není jisté, zda bude vůbec vytvořen, ale jeho konstrukce představuje náročnou operaci vyžadující mnoho systémových prostředků.

Jednoduchým generickým řešením je vytvořit šablonovou třídu, která se chová jako ukazatel na daný objekt, avšak nevytváří jej, dokud k němu není přistoupeno pomocí přetíženého operátoru dereference nebo přístupu k členskému atributu (operátory '`*`' a '`->`'). Taková proxy si uloží parametry konstruktoru pro cílový objekt do n-tice a samotný konstruktor volá až v případě potřeby pomocí funkce `call` ze strany 34.

Představu o implementaci takové proxy je možné si udělat z šablony `LazyProxy` v přiložených kódech. Dále vytvoříme jednoduchou obalovací funkci `make_lazy_proxy`, která nám usnadní vytvoření proxy s odloženou konstrukcí objektu pomocí dedukce šablonových parametrů.

Následující ukázka demonstruje vytvoření „ukazatele“ na obrázek, který však není načten do paměti, dokud není volána nějaká metoda příslušného objektu.

Ukázka 4.42: Použití proxy s odloženou konstrukcí objektu

```
1 auto img = make_lazy_proxy<Image>("very_large_image.png"); // LazyProxy
2 /* ... do stuff ... */
3 if (something_unlikely_happened)
4     img->show();  // image gets loaded here if something unlikely happened
```

4.4.3 Abstraktní továrna

Návrhový vzor *abstraktní továrna* nalezne své využití v situaci, kdy potřebujeme generovat objekty několika typů tříd konzistentně. Vlastně vytváří abstrakci nad rodinou továren pro konkrétní případy.

Ukážeme si to na příkladu imaginární závodní jídelny, která v rámci úsporných opatření vyrábí každý den polévku, hlavní jídlo i dezert ze stejných surovin. Menu v pondělí tedy je gulášová polévka, guláš a jako dezert párek, v úterý pak zelná polévka, vepřo-knedlo-zelo a zelný salát a tak dále.

Abstraktní továrna tedy musí obsahovat tři jednotky, nazvěme je „výrobní linky“, a to na polévku, hlavní chod a dezert. Obecnou definici takové výrobní linky bez návaznosti na náš příklad ukazuje výpis 4.43. Šablona `TypeId` zde slouží pouze k rozpoznání jednotlivých přetížených verzí funkce `do_create`.

Ukázka 4.43: Abstraktní výrobní linka

```
1 template <class T>
2 struct AbstractFactoryFlowLine {
3     virtual ~AbstractFactoryFlowLine() {}
4     virtual T* do_create(TypeId<T>) = 0;
5 };
```

Tím máme vyřešeno rozhraní, ne však implementaci. Jedním z možných postupů je instance vytvářet pomocí operátoru new. Následující šablona bere jako parametr seznam o dvou položkách. První z nich představuje abstraktní bázovou třídu pro náš produkt, druhý již konkrétní typ k vytvoření. Další parametr určuje předka této linky. Z tohoto rozhraní je již možné vytušit, že budeme jednotlivé výrobní linky v továrně generovat jako lineární hierarchii (strana 32).

Ukázka 4.44: Výrobní linka pomocí operátor new

```
1 template <class AbsT, class ConcT, class Parent>
2 struct FactoryNewFlowLine<List<AbsT, ConcT>, Parent> : public Parent {
3     typedef AbsT AbstractProduct;
4     typedef ConcT Product;
5     Product* do_create(TypeId<AbstractProduct>) { return new Product; };
6 };
```

Když máme komponenty pro jednotlivé výrobní linky, můžeme z nich směle poskládat továrnu. Abstraktní továrna využije generování izolovaných hierarchií, čímž zdědí od výrobních linek všech typů, které má produkovat. Zde přijde na scénu již mnohokrát použitý seznam typů. Bude vystupovat jako první parametr. Pro každý typ abstraktní továrna vygeneruje výrobní linku a od všech následně zdědí.

Metoda `do_create` jednotlivých linek je spíše implementační záležitostí, uživatel bude pracovat s metodou `create` parametrizovanou požadovaným typem. Volání pak může vypadat například následovně: `factory->create<Polevka>()`. To vytvoří gulášovou či zelnou polévku podle dané konkrétní továrny implementující abstraktní rozhraní.

Vnořená šablona `Factory` se již stará přímo o generování konkrétní továrny. Jako parametr bere přirozeně seznam konkrétních typů, které budou generovány. Tyto typy musí být na příslušných pozicích k jejich rodičovskému typu v seznamu typů pro abstraktní továrnu. Poté tyto informace použije k vygenerování lineární hierarchie výrobních linek.

Vnořená šablona má k dispozici šablonové parametry nadřazené šablony. Může tedy vzít jako prvky dvouprvkového seznamu seznam abstraktních typů a seznam konkrétních typů.

Jedná se tedy o seznam dvou seznamů. Pomocí operace transpozice se z něj vyrobí seznam dvojic [abstraktní typ, konkrétní typ]. Tyto dvojice již akceptuje jako parametr naše šablona výrobní linky. Na vršek této hierarchie postavíme samotnou abstraktní továrnu. Třída konkrétní továrny nepotřebuje žádné další metody, všechny potřebné zdědila od výrobních linek a abstraktní továrny.

Ukázka 4.45: Generická abstraktní továrna

```
template <class Ts, class Unit = Tmpl<AbstractFactoryFlowLine>>
class AbstractFactory : public Inherit<typename ForEach<Unit, Ts>::type> {
public:
    typedef typename Ts::type AbstractProducts;

    template <typename T> T* create() {
        typename MetaCall<Unit, List<T>>::type* flow_line = this;
        return flow_line->do_create(TypeId<T>());
    }

    template <class Types, class CreateUnit = Tmpl<FactoryNewFlowLine>>
    class Factory : public LinearInherit<CreateUnit,
        typename Transpose<List<Ts,Types>>::type, AbstractFactory<Ts,Unit>>
    {
    public:
        typedef AbstractFactory<Ts, Unit> AbstractFactoryType;
        typedef typename Ts::type AbstractProducts;
        typedef typename Types::type Products;
    };
};
```

V momentě, kdy máme k dispozici výše uvedené šablony, se po vložení správného hlavičkového souboru náš problém generování jídel v simulované závodní jídelně zredukuje na pár řádků obsahujících výčet typů.

Ukázka 4.46: Vygenerování abstraktní továrny

```
typedef AbstractFactory<List<Polevka,HlavniChod,Zakusek>> Jidelna;
typedef Jidelna::Factory<List<GulasPolevka,Gulas,Parek>> MenuPondeli;
typedef Jidelna::Factory<List<ZeliPolevka,VeproKnedlo,ZeliSalat>> MenuUtery;
```

Policy-based design Operátor `new` není jediný způsob vytváření objektů. Můžeme využít jiný již existující kód nebo klonovat předlohy. To reflektuje volitelná možnost změny šablony pro výrobní linku. Uživatel tak může upravit její chování (krom možnosti jej upravit selektivně pro některé typy pomocí specializace). Tomuto přístupu se říká návrh založený na zásadách (*policy-based design*).

Východiskem je teze, že jedna verze objektu není vhodná pro všechny situace a je dobré mít možnost parametrizovat některé vlastnosti jeho návrhu. V konkrétní situaci se pak na požádání vygeneruje verze, která se pro daný účel hodí nejlépe.

Alexandrescu [2] ukazuje pěkný příklad využití tohoto idiomu pro implementaci generického chytrého ukazatele. Existuje několik modelů držení odkazovaného objektu, například výlučné vlastnictví, sdílené s počítáním referencí, sdílené nekontrolované („hloupý ukazatel“) a podobně. Stejně tak existuje několik přístupů ke kontrole pokusu o dereferenci nulového ukazatele – aserce, výjimka či bez kontroly. Taky můžeme operace s ukazatelem různě synchronizovat mezi vlákny a tak dále a tak podobně. Policy-based design umožňuje z relativně nízkého počtu šablon generovat všechny možné kombinace chování objektu.

4.4.4 Další vzory

Další návrhové vzory pouze letmo naťukneme. Nejsou ani součástí přiložených kódů, protože na jejich knihovní implementaci se dá narazit na každém kroku.

Příkaz Generický obal pro všemožné volatelné objekty, ať už se jedná o funkční objekt nebo ukazatel na funkci. Samotná instance této šablony bývá zpravidla sama funkční třídou. Ve standardní knihovně (přesněji TR1) je k nalezení pod názvem `std::function`.

Šablona `bind` z knihoven Boost dokonce umí transparentně vygenerovat funkční objekt volající funkci s napevno svázanými parametry s danými hodnotami. Taktéž umožňuje měnit pořadí parametrů.

Pozorovatel Vzor, kdy objekt spravuje seznam dalších objektů, které jsou eventuálně upozorněny na změnu jeho stavu a mohou na ni adekvátně reagovat. V C++ existuje několik implementací tohoto vzoru založených na principu signal/slot. Jmenujme například znovu knihovnu Boost, konkrétně *Boost.Signal*, kde je signál současně i funkčním objektem.

4.5 Šablony výrazů

Jeden ze způsobů jak krásně zjednodušit tvorbu instancí i velmi složitých šablon poskytují přetížené operátory. Protože to vlastně jsou funkce, mají také schopnost dedukce šablonových parametrů podle typů svých operandů.

Například *Boost.Spirit* dokáže pomocí operátorů v C++ kódu soustavou šablon vygenerovat celý parser. Taktéž knihovna *Blitz++* používá tuto techniku k optimalizaci operací s vektory a s maticemi.

Odložené vyhodnocení výrazů Naším cílem bude napsat sadu generických šablon, které vývojáři umožní zadat v jazyce C++ matematický výraz. Ten však nebude vyhodnocen hned, nýbrž může být uložen do proměnné či předán funkci jako parametr a vyhodnocení se provede až na požádání.

Systém bude možno použít s kterýmkoli datovým typem podporujícím použité operace (například `operator+`). U proměnných, které explicitně označíme pomocí šablonové funkce `Ref`, se při každém vyhodnocení bude brát jejich aktuální hodnota. Musíme si tedy dát pozor na jejich životnost. Vyhodnocení výrazu spustíme pomocí operátoru volání funkce.

Ukázka 4.47: Použití odloženého vyhodnocení výrazů

```
1 int x;
2 auto expr = Ref(x) * (Ref(x) - 1);
3 x = 3; expr(); // expr() returns 6
4 x = 5; expr(); // expr() returns 20
5 std::function<int()> f(expr); // type erasure
```

Výsledný datový typ proměnné `expr` nepůsobí příliš přátelsky[2], přestože výraz není z nejsložitějších. Proto, a také z důvodu lepší binární kompatibility, se občas používá technika zvaná *type erasure* (mazání typu). Pro tyto účely dobře poslouží nová šablona ze standardní knihovny `function`. Datový typ funkčního objektu `f` je již o poznání přívětivější.

[2] ExprMul<ExprRef<int>, ExprSub<ExprRef<int>, ExprValue<int>>>

Kapitola 5

Testování překladačů

Implementace vlastností nové normy do překladačů právě podstupuje překotný vývoj. Tento oddíl se zaměřuje na získání hrubého přehledu o aktuálním stavu podpory nových vlastností v různých kompilátorech formou sady jednoduchých testů.

5.1 Metodika

Pro samotné testování byl v rámci zachování nezávislosti na platformě a snahy o nepřidávání dalších závislostí pro sadu přiložených demonstračních kódů zvolen jednoduchý skript v prostředí buildsystému CMake[1]. Ten postupně spouští jednotlivé soubory s příponou `cpp` z adresáře `test` a zaznamenává výstup překladu i běhu programu. Také průběžně vytváří soubor se souhrnnými výsledky, ve kterém ke každému testu přiřadí jeho výsledek. Výsledkem testu může být jedna ze tří hodnot:

- pass (P) – Překladač úspěšně přeložil test a tento se chová dle očekávání.
- compilation failure (CF) – Překladač nezvládl zdrojový soubor přeložit. Pravděpodobně tedy testovanou vlastnost nepodporuje.
- runtime failure (RF) – Překladač zdrojový soubor přeložil, avšak test po spuštění vrací špatné výsledky. To značí chybu v implementaci testované vlastnosti.

Nutno podotknout, že testy mají umožnit získat hrubý přehled o podpoře šablon a nových vlastností C++0x. Zdaleka nepokrývají všechny možné i nemožné stavy a kombinace syntaktických a sémantických konstrukcí. Neodhalují tedy možné interference vzniklé použitím více vlastností jazyka, to však ani není jejich účelem.

Příklady, které jsou součástí hlavní části práce byly vyvíjeny a testovány pomocí překladače GCC verze 4.4.3. Většina z nich by neměla činit problém kterékoli verzi zmíněného překladače řady 4.4 a výše. Tyto příklady také nejsou pro naše testovací účely vhodné, protože stačí, aby testovaný překladač nezvládal šablony s proměnným počtem parametrů nebo úhlové závorky případně některou další hojně využívanou vlastnost a okamžitě by drtivá většina testů dopadla selháním překladu. Testy se tedy raději zaměřují na jednotlivé vlastnosti pokud možno izolovaně.

[1] http://www.cmake.org

5.2 Výsledky testů

Zde jsou prezentovány výsledky vybraných testů pro překladače GCC[2] a MSVC[3]. Konkrétní test lze snadno identifikovat podle čísla – název souboru s daným testem začíná právě tímto číslem. Testy prověřující implementace vlastností nové normy začínají číslem 020.

Číslo testu	**Stručný popis**	**GCC** 4.4.3	**GCC** 4.5.0	**MSVC** 2010
001	jednoduchá šablona + specializace	P	P	P
010	parametr šablony závislý na jiném parametru	P	P	P
012	šablonový parametr šablony	P	P	P
020	constexpr proměnná	CF	P	CF
021	constexpr funkce	CF	CF	CF
030	klíčové slovo auto	P	P	P
040	nová syntaxe hlavičky funkcí	P	P	P
050	decltype	P	P	P
060	úhlová závorka	P	P	P
070	variadické šablony – funkce	P	P	CF
071	variadické šablony – třída	P	P	CF
073	variadické šablony – sizeof	P	P	CF
090	řetězcový uživatelský literál	CF	CF	CF
091	šablonový uživatelský literál	CF	CF	CF
100	statická aserce	P	P	P
110	rvalue reference	P	P	P
120	nullptr	CF	CF	P
130	smyčka for založená na rozsahu	CF	CF	CF
140	initializer_list + funkce	P	P	CF
141	initializer_list + std::vector	P	P	CF
147	sjednocená inicializace	P	P	CF
150	lambda funkce	CF	P	P
151	lambda se zachováním kontextu	CF	P	P
152	lambda jako funkční objekt v proměnné	CF	P	P

5.3 Vyhodnocení

Jak je vidět, testované překladače úspěšně konvergují ke stavu plné podpory nového standardu. Aktuální verze GCC v testovaných vlastnostech zaostává za produktem Microsoftu pouze v implementaci typově bezpečného nulového ukazatele, což však je již podle posledních zpráv ve vývojových repositářích napraveno. Naproti tomu chybějící variadické šablony u Visual C++ bohužel znemožňují kompilaci většiny příkladů uvedených v hlavní části práce.

Překladače se tedy podpoře normy přibližují. Dokládá to i posun mezi dvěma verzemi GCC. Situace se zlepšila i u mnoha testů, které nejsou zahrnuty v tabulce. Lze tedy očekávat, že zanedlouho pozbudou výše uvedené výsledky testů významu, protože nové verze kompilátorů na tom budou velmi pravděpodobně ještě mnohem lépe.

[2]GNU Compiler Collection (http://gcc.gnu.org/)
[3]Microsoft Visual C++ (http://msdn.microsoft.com/en-us/visualc/default.aspx)

Kapitola 6

Závěr

Na příkladech jsme ukázali široké možnosti šablonového metaprogramování. Jednalo se spíše o průřezový přehled, který ani v nejmenším nepostihl celou šíři možných technik a jejich využití. Taktéž by se i u příkladů zahrnutých v tomto výčtu dalo jít více do hloubky, podrobněji diskutovat jejich silné stránky i slabiny.

Byly demonstrovány nové možnosti, které přináší nadcházející verze normy jazyka C++ v oblasti metaprogramování. Jedná se především o podporu šablon s proměnným počtem parametrů a do budoucna případné zavedení konceptů, to vše doplněno dalšími změnami, které, po odmyšlení nutnosti zpětné kompatibility sahající až k jazyku C, působí velmi konzistentním dojmem.

Dalším z cílů této práce bylo představit šablony jazyka C++ jako plnohodnotný programovací jazyk, ve své doméně výpočetně úplný. S těmito znalostmi jsme byli schopni generovat staticky polymorfní kód a obecně implementovat některé návrhové vzory.

Přestože se jednalo o příklady demonstrační, mnohé z nich by bylo možno ihned použít ve skutečných softwarových projektech (některé s drobnými úpravami či doplněním). To také vyplývá ze skutečnosti, že se jednalo o ukázku psaní kódu, který je generický a znovupoužitelný. Příkladů by zajisté bylo možno vymyslet více, nicméně doufám, že tato práce poskytla dobrý základ pro vytvoření představy o rozmanitosti a rozsahu světa metaprogramování pomocí šablon.

Literatura

[1] ABRAHAMS, David, GURTOVOY, Aleksey. *A Deeper Look at Metafunctions*. [online]. 2004, [cit. 15. 4. 2010].
URL <http://www.artima.com/cppsource/metafunctions.html>

[2] ALEXANDRESCU, Andrei. *Moderní programování v C++*. Computer Press. 2004. ISBN 80-251-0370-6.

[3] BECKER, Thomas. *C++ Rvalue References Explained*. [online]. 2009, [cit. 26. 4. 2010].
URL <http://thbecker.net/articles/rvalue_references/section_01.html>

[4] BRIGHT, Walter. *Templates Revisited*. [online], [cit. 18. 4. 2010].
URL <http://www.digitalmars.com/d/2.0/templates-revisited.html>

[5] ISO/IEC JTC1 SC22 WG21 N 3092. Technická zpráva. 2010. Final draft.
URL <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf>

[6] ELLIS, Margaret A., STROUSTRUP, Bjarne. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 1990. ISBN 0-201-51459-1.

[7] GREGOR, Douglas, SIEK, Jeremy, GARCIA, Ronald, et al. *Concepts for C++0x*. [online]. 2005, [cit. 22. 4. 2010]. 1. revize.
URL <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1849.pdf>

[8] GREGOR, Douglas, JÄRVI, Jaakko. Variadic templates for C++. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, (s. 1101–1108). ACM, New York, NY, USA. 2007. ISBN 1-59593-480-4.

[9] HUTCHINGS, Ben. *C++ Templates FAQ*. [online]. 2006, [cit. 26. 4. 2010].
URL <http://womble.decadentplace.org.uk/c++/template-faq.html>

[10] KICZALES, Gregor, LAMPING, John, MENDHEKAR, Anurag, et al. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP)*. 1997.

[11] KOSKINEN, Johannes. Metaprogramming in C++. Technická zpráva. 2004.
URL <http://www.cs.tut.fi/~kk/webstuff/MetaprogrammingCpp.pdf>

[12] MAHMOUD, Qusay H. *Using and Programming Generics in J2SE 5.0*. [online]. 2004, [cit. 15. 4. 2010].
URL <http://java.sun.com/developer/technicalArticles/J2SE/generics/>

[13] MCKEEMAN, William M., HORNING, James J., WORTMAN, David B. *A Compiler Generator*. 1971. ISBN 13-155077-2.

[14] MEYERS, Randy. *The New C: X Macros*. [online]. 2001, [cit. 18. 4. 2010]. URL <http://www.drdobbs.com/cpp/184401387>

[15] PRATA, Stephen. *Mistrovství v C++*. Computer Press. 2004. ISBN 8025100987. 2. aktualizované vydání.

[16] *rpcgen Programming Guide*. [online], [cit. 18. 4. 2010]. URL <http://docs.freebsd.org/44doc/psd/22.rpcgen/paper.pdf>

[17] SPINCZYK, Olaf, GAL, Andreas, SCHRÖDER-PREIKSCHAT, Wolfgang. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, (s. 53–60). Australian Computer Society, Inc., Darlinghurst, Australia, Australia. 2002. ISBN 0-909925-88-7.

[18] STROUSTRUP, Bjarne. A history of C++: 1979–1991. (s. 699–769). 1996.

[19] STROUSTRUP, Bjarne. *The C++ Programming Language: Special Edition*. Addison-Wesley Professional, 3 ed. 2000. ISBN 0-201-70073-5.

[20] STROUSTRUP, Bjarne. Evolving a language in and for the real world: C++ 1991-2006. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, New York, NY, USA. 2007. ISBN 978-1-59593-766-X.

[21] STROUSTRUP, Bjarne. *The C++0x „Remove Concepts" Decision*. [online]. 2009, [cit. 27. 4. 2010]. URL <http://www.drdobbs.com/cpp/218600111>

[22] STROUSTRUP, Bjarne. *What is C++0x*. [online]. 2009. URL <http://www2.research.att.com/~bs/what-is-2009.pdf>

[23] VANDEVOORDE, David, JOSUTTIS, Nicolai M. *C++ Templates: The Complete Guide*. Addison-Wesley Professional. 2002. ISBN 0201734842.

[24] VANDEVOORDE, Daveed. *Modules in C++*. [online]. 2007, [cit. 22. 4. 2010]. 5. revize. URL <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2316.pdf>

[25] VELDHUIZEN, Todd L. C++ Templates are Turing Complete. Technická zpráva, Indiana University Computer Science. 2003.

[26] WIKIPEDIA. *Metaprogramming — Wikipedia, The Free Encyclopedia*. [online]. 2010, [cit. 20. 4. 2010]. URL <http://en.wikipedia.org/wiki/Metaprogramming>

Ukázky kódu

Rejstřík